VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY

Computer Network Lab (CO3094)

Assignment 1 Report

VIDEO STREAMING APPLICATION

Lecturer.	Nguyễn Lê Duy Lai
Class :	CC05-CC01
Students:	Cao Bá Huy - 1952713 (CC05)
	Huỳnh Phước Thiện - 1952463 (CC01)
	Đỗ Đăng Khoa - 1952295 (CC01)

HO CHI MINH CITY, SEPTEMBER 2021

Contents

1 Introduction

The Real-Time Streaming Protocol (RTSP) establishes and controls either a single or several time-synchronized streams of continuous media such as audio and video. It does not typically deliver the continuous streams itself, although interleaving of the continuous media stream with the control stream is possible. In other words, RTSP acts as a "network remote control" for multimedia servers.

When a user or application attempts to stream video from a remote source, the client device sends an RTSP request to the server to determine the options, such as set up, pause, play, and tear down (stop). The server then returns reply to confirm that if the request is valid or not. Once the client makes a request, it transmits a media description request to the streaming server, and the server responds with a description of the media. First of all, the client sends a setup request and the server responds with information about the transport mechanism. Once the setup process is completed, the client initiates the streaming process by telling the server to send the bitstream – a binary sequence – using the transport mechanism specified in the setup request.

Our report aims the process of implementing a simple Video streaming application using Real-Time Streaming Protocol (RTSP). We will illustrate this process step by step by analysing the requirements, the used functions, the basic components of the app, the dataflow model, the class diagram, the user manual and most importantly the implementation code.

2 Analysis of problem requirements

2.1 Source files

Building this video application will require 6 source files :

1. Client.py
2. ClientLauncher.py
3. RtpPacket.py
4. Server.py
5. ServerWorker.py
6. VideoStream.py

2.2 Functional requirements

Implementing a streaming video server and client that communicate using the Real-Time Streaming Protocol (RTSP) and send data using the Real-time Transfer Protocol (RTP).

2.3 Non-functional requirements

1. The standard RTSP port is 554, but you will need to choose a port number greater than 1024.
2. For the SETUP process to the server, when creating the datagram socket for receiving RTP data, we have to set the timeout on the socket to 0.5 seconds.
3. When the server receives the PLAY-request from the client, the server reads one video frame from the file and creates a RtpPacket-object and then sends the frame to the client over UDP every 50 milliseconds.
4. The server streams a video is encoded into a proprietary MJPEG file format.
5. The application should be able to run on PC.

2.4 Tasks

We will need to implement :

- The RTSP protocol in the client (named as Client.py).
- The RTP packetization in the server (named as RtpPacket.py).
- The customized interactive player for displaying the transmitted video.

3 Description of different functions

In this section, we will give you a brief descriptions about the functions in each of the file.

3.1 VideoStream.py

The main function of this file is to open an read the video file in form of byte stream. Moreover, it is also used to read a frame of the video and return the frame number for tracking purposes. The table below describes the the functions in this file :

Function	**Description**
init(self, filename)	A constructor to open and read the file in byte stream
nextFrame(self)	Get the frame from the first 5 bits, increase the frame number by 1 and return the frame.
frameNbr(self)	Return the frame number

3.2 ServerWorker.py

When ever a new client comes, a new thread is started to process the requests sent from Client. The request includes the request type, required file name,... Server-Worker also response to Client by sending the needed video one frame by one and reply to determine if the request is valid or not. The table below describes the the functions in this file :

Function	**Description**
init(self, clientInfo)	A constructor to store the client information.
run(self)	This function is used to start and run the server.
recvRtspRequest(self)	Receive the RTP request form the client throught the socket.
processRtspRequest(self, data)	Process the RTP request based on the option from the request (SETUP, PLAY , PAUSE , TEARDOWN).
makeRtp(self, payload, frameNbr)	Packet the data into RTP packet through UDP transmission protocol .
sendRtp(self)	Send the RTP packet
replyRtsp(self, code, seq)	Reply the to the client.

3.3 Server.py

This file containing the server creates a socket and receives the the client information. It will forward the data to the ServerWorker.py to handle the request processing.

3.4 RtpPacket.py

The main function of this file is to demonstrate the mechanism to encode the variables of a RTP packet such as version, padding,extension, contributing sources (cc), sequence number , marker, payload type(pt), source identifier(ssrc) into a RTP packet and how to decode it. Moreover, it also provides the packet payload and the packet itself. The table below describes the the functions in this file :

Function	**Description**
init(self)	A constructor to assign the header in byte stream.
encode(self, version, padding, extension, cc, seqnum, marker, pt, ssrc, payload)	Encode the RTP packet with header fields and payload.
decode(self, byteStream)	Decode the RTP packet.
version(self)	Return the RTP version.
seqNum(self)	Return the sequence number corresponding to the frame number
timestamp(self)	Return the time stamp.
payloadType(self)	Return the type of the payload.
getPayload(self)	Return the Payload.
getPacket(self)	Return the RTP packet

3.5 Client.py

The main function of this file is to set up the graphical user interface including 4 buttons (SETUP, PLAY,PAUSE and TEARDOWN). After that, it will connect to the server through the created socket and can begin to send the request to the server. The table below describes the the functions in this file :

Function	Description
init(self, master, serveraddr, serverport, rtpport, filename)	The constructor to assign some basic information.
createWidgets(self)	The function to create the GUI.
setupMovie(self)	Send the set up request to the server when the user clicks the SETUP button.
exitClient(self)	Send the tear down request to the server.
pauseMovie(self)	Send the pause request to the server.
playMovie(self)	Send the play request to the server.
listenRtp(self)	Listen to RTP packets and handle them.
writeFrame(self, data)	Write the received frame to a tempoary image file and return the image file.
updateMovie(self, imageFile)	Update the image file as video frame in the to the user interface.
connectToServer(self)	Connect to the Server through a socket and start a new RTSP/TCP session.
sendRtspRequest(self, requestCode)	Send the RTP request to the server.
recvRtspReply(self)	Receive the RTSP reply from the server.
parseRtspReply(self, data)	Analyzing the RTSP reply from the server.
openRtpPort(self)	Open RTP socket binded to a specified port.
handler(self)	Handler on explicitly closing the GUI window.

3.6 ClientLauncher.py

This function is used to get the server name, server port , the client RTP port and the video file name to initiate the application.

4 The list of components

When the user clicks on the buttons on the user interface, we will need to implement the actions for the respective request types. As the client starts, it also opens the RTSP socket to the server and uses this socket for sending all RTSP requests. Overall, we have 4 main requests from the client :

1. SETUP
2. PLAY
3. PAUSE
4. TEARDOWN

For each of the requests from the client, the server always replies to all the messages the client sends. The code 200 means that the request was successful while the codes 404 and 500 represent $FILE\_NOT\_FOUND$ error and connection error respectively. We will now discuss about the functions that will be performed when the client send 4 different requests.

4.1 SETUP request

The SETUP request specifies the transport mechanism to be used for the streamed media. The request also contains the video file and a transport specifier. This specifier typically includes a local port for receiving RTP data (audio or video). When the server receives the request, if the received data (the video file) is valid, the server will generate a randomized RTSP session ID and map it to the client information dictionary. Afterward, the server will send a reply to confirm the successful status. Below is the code for the processing the SETUP request.

```
    # Process SETUP request
    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])

            # 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]
```

Listing 1: The Set up request processing from the server in ServerWorker.py

This is an example of the SETUP request form the client :

```
SETUP movie.Mjpeg RTSP/1.0
CSeq: 1
Transport: RTP/UDP; client_port= 25000
```

Where :

- SETUP is the option.
- movie.Mjpeg is the encoded video file.
- RTSP/1.0 is the Real time streaming protocol version 1.0 .
- CSeq is the sequence number.
- RTP/UDP is the transmission protocol.
- 25000 is the port that will be connected.

After the client the request, the server will send a reply :

```
RTSP/1.0 200 OK
CSeq: 1
Session: 123456
```

Where:

- 200 means successful.
- Session : 123456 is the session ID for the client.

4.2 PLAY request

The PLAY method tells the server to start sending data via the mechanism specified in SETUP. A client MUST NOT issue a PLAY request until any outstanding SETUP requests have been acknowledged as successful.A range can be specified. If no range is specified, the stream is played from the beginning and plays to the end, or, if the stream is paused, it is resumed at the point it was paused.
When the server receives the PLAY request from the client, the server will create a Socket to start transmitting the video packet.

```
      # Process PLAY request
      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()
```

Listing 2: The Play request processing from the server in ServerWorker.py

After that this function will called to a function called sendRtp in the same class to start sending RTP packets to the client.

```
   def sendRtp(self):
      """Send RTP packets over UDP."""
      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")
```

Listing 3: Sending RTP packets

In this sendRtp function, the functions from VideoStream.py which are nextFrame() and frameNbr() are used to get the frames and their corresponding frame number from the video file :

```python
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
```

Listing 4: The VideoStream class used to get the video frames and frame numbers

Another important component of the PLAY function is the RTP packets. When the server receives the PLAY-request from the client, the server reads one video frame from the file and creates a RtpPacket-object which is the RTP-encapsulation of the video frame. For the encapsulation, the server calls the encode function of the RtpPacket class named encode to set the header for the RTP packets. How it will do it will be discussed later in the implementation section.

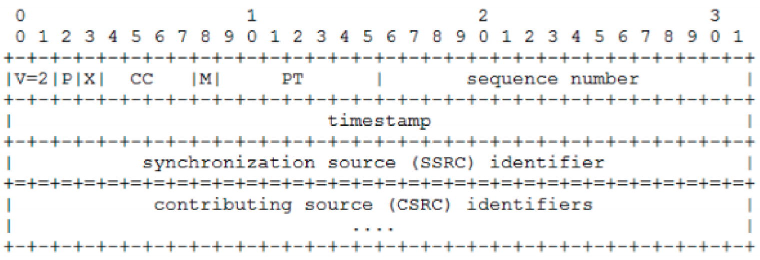


Figure 1: The RTP packet header format.

```
PLAY movie.Mjpeg RTSP/1.0
CSeq: 2
Session: 123456
```

Listing 5: The PLAY request from the client

4.3 PAUSE request

The PAUSE request is used to tell the client to stop sending getting frames and send RTP packets to the client. Note that when receiving the PAUSE request, the server is not torn down but instead the state will be set to READY and wait for the next request from the client.

```
      # Process PAUSE request
      elif requestType == self.PAUSE:
         if self.state == self.PLAYING:
            print("processing PAUSE\n")
            self.state = self.READY

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

            self.replyRtsp(self.OK_200, seq[1])
```

Listing 6: The PAUSE request processing from the server.

```
PAUSE movie.Mjpeg RTSP/1.0
CSeq: 3
Session: 123456
```

Listing 7: The PAUSE request sent by the client.

4.4 TEARDOWN request

The TEARDOWN request is used to close the RTP socket between the client and the server. After that, the state will be set to INIT.

```
      # Process TEARDOWN request
      elif requestType == self.TEARDOWN:
         print("processing TEARDOWN\n")

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

         self.replyRtsp(self.OK_200, seq[1])
```

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

Listing 8: The PAUSE request processing from the server.

```
TEARDOWN movie.Mjpeg RTSP/1.0
CSeq: 5
Session: 123456
```

Listing 9: The TEARDOWN request sent from the client

5 Model and dataflow

The dataflow describes the flow of the data between external entities and data points. External entities are objects that sends or receives data, exchanging information between each other and other data points. They are the sources and destinations of information entering or leaving the system. A process receives an input and produces an output and forward them to either an entity or a data point. Data points are storage and can range from databases to simple files. They mainly store data to be used by processes.
The notation we have chosen to use in our dataflow includes: squares for external entities, rounded square split into 2 fields for process with an ID number for easier reference, and quadrilaterals made up of 2 rectangles with a missing leftmost edge for data points. Lastly, arrows indicate the flow of data.
The procedure goes as follows: The user starts by launching the server and a client, providing startup information such as port to be used, media file to be sent, ... Process 1 and 2 will create an instance of a server and client based on the arguments provided. The user may then choose to press the buttons that appeared on the client, indicated as Process 3, prompting the client to execute the functions binded to those buttons, whose end result is usually Process 4 taking place. It processes the RTSP request message sent by the client and tells the server what to do with it. The server responds with its own RTSP reply, and in some cases, a packetized frame. Before that, it will need to retrieve said frame from a video stream object, triggering Process 5. Process 5 can also take requests asking for the frame number the video stream object is currently at. Process 5 will then return said result to the server. The frame will be converted to a payload of a packet in Process 6, and sent off to the client. All that is left is for the client to unpacket it and show the user the original frame with Process 7.

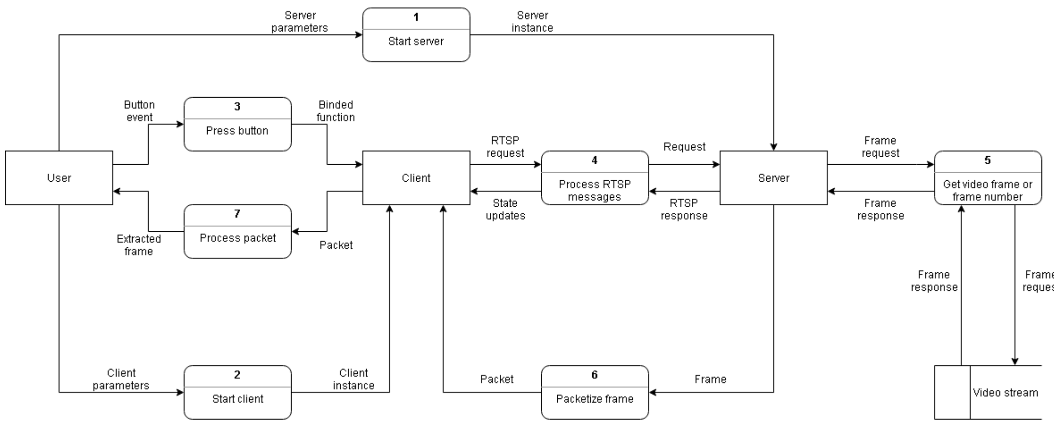


Figure 2: Dataflow diagram of the system

6 Class diagram

The class diagram describes the content of classes and their functionalities in the form of member attributes and methods. It is also used to show the connection and hierarchy between classes where classes may employ many objects of other classes.
In our particular case, there are 5 classes present within the system. The RtpPacket class is perhaps the most basic but fundamental class. It is used in both the Client class ServerWorker class when sending and receiving rtp packets.
The Client class is responsible for most of the user's actions, storing information regarding the RTSP/UDP connection and video progress. An instance of the Client class is connected to an instance of the Server class when first initiating a connection and is relegated to be handled by an instance of the ServerWorker class for the rest of its lifespan.
The Server class manages incoming connections and creates ServerWorker threads dedicated to interacting with the clients issuing those connections in the future.
The ServerWorker's main responsibility is responding to its client's requests. It can either send RTSP replies or the RTP packets themselves. It also uses an instance of a VideoStream class to manage the information about the frames in the video it is supposed to send.
The VideoStream class manages the progress of the video, split into frames.

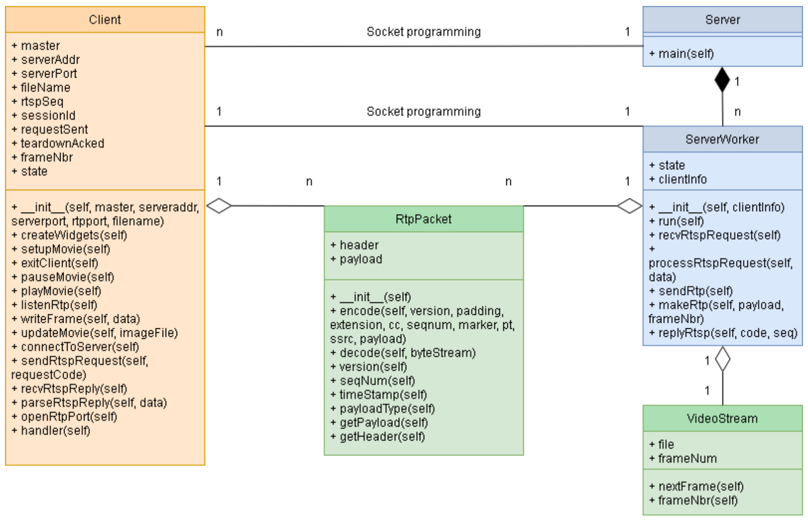
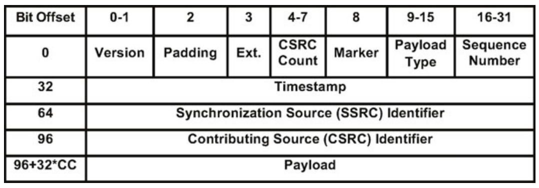


Figure 3: The class diagram of the system

7 Implementation

7.1 RtpPacket.py

In this file, our task is to implement the encode function of the RtpPacket class to set the header using the bitwise operations. The picture below shows the number of bits needed for each field and we will base on that to set the bits for the header.



Bit Offset	0-1	2	3	4-7	8	9-15	16-31
0	Version	Padding	Ext.	CSRC Count	Marker	Payload Type	Sequence Number
32	Timestamp						
64	Synchronization Source (SSRC) Identifier						
96	Contributing Source (CSRC) Identifier						
96+32*CC	Payload						

Figure 4: The numebr of bits for each fields

And here is the implementation code :

```python
def encode(self, version, padding, extension, cc, seqnum, marker, pt, ssrc,
 payload):
   """Encode the RTP packet with header fields and payload."""

   timestamp = int(time())
   self.header = bytearray(HEADER_SIZE)


   header = bytearray(HEADER_SIZE)
   header[0] = (header[0]|version<<6)&0xC0
   header[0] = (header[0] | padding << 5)
   header[0] = (header[0] | extension << 4)
   header[0] = (header[0] | (cc & 0x0F))

   header[1] = (header[1] | marker << 7)
   header[1] = (header[1] | (pt & 0x7f))

   header[2] = (seqnum & 0xFF00) >> 8
   header[3] = (seqnum & 0xFF)
   header[4] = (timestamp >> 24)
   header[5] = (timestamp >> 16) & 0xFF
   header[6] = (timestamp >> 8) & 0xFF
   header[7] = (timestamp & 0xFF)
   header[8] = (ssrc >> 24)
   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 = ...
        self.payload = payload
```

Listing 10: The encode function

7.2 Client.py

In this file, we will implement the necessary functions to connect with the server, send request to the server, receive request from the server and show the frame images on the GUI window. We will first implement the Send RTP request function which is used to send the request from the client to the customer, the client will send a request which specifies the option (SETUP, PLAY , PAUSE , TEARDOWN) and some other information such as the : transmission protocol, the sequence number , the RTSP version and the client port number.

```
    def connectToServer(self):
        """Connect to the Server. Start a new RTSP/TCP session."""
        self.rtspSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        try:
            self.rtspSocket.connect((self.serverAddr, self.serverPort))
        except:
            tkMessageBox.showwarning('Connection Failed', 'Connection to \'%s\'
     failed.' %self.serverAddr)
```

The function used to connect 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

            # Create the RTP request.
            request = "SETUP " + str(self.fileName) + " RTSP/1.0\nCSeq: " + str(
     self.rtspSeq) + "\nTransport: RTP/UDP; client_port= " + str(self.rtpPort)
            self.rtspSocket.send(request.encode("utf-8"))

            # Keep track of the sent request.
            self.requestSent = self.SETUP

        # Play request
        elif requestCode == self.PLAY and self.state == self.READY:
            # Update RTSP sequence number.
            self.rtspSeq += 1

            # Write the RTSP request to be sent.
            request = "PLAY " + str(self.fileName) + " RTSP/1.0\nCSeq: " + str(
     self.rtspSeq) +"\nSession: " + str(self.sessionId)
            self.rtspSocket.send(request.encode("utf-8"))
            # Keep track of the sent request.
            self.requestSent = self.PLAY
        # Pause request
        elif requestCode == self.PAUSE and self.state == self.PLAYING:
            # Update RTSP sequence number.
            self.rtspSeq += 1

            # Write the RTSP request to be sent.
            request = "PAUSE " + str(self.fileName) + " RTSP/1.0\nCSeq: " + str(
     self.rtspSeq) + "\nSession: " + str(self.sessionId)
            self.rtspSocket.send(request.encode("utf-8"))

            # Keep track of the sent request.
            self.requestSent = self.PAUSE
        # Teardown request
        elif requestCode == self.TEARDOWN and not self.state == self.INIT:
            # Update RTSP sequence number.
            self.rtspSeq += 1

            # Write the RTSP request to be sent.
            request = "TEARDOWN " + str(self.fileName) + " RTSP/1.0\nCSeq: " + str
     (self.rtspSeq) + "\nSession: " + str(self.sessionId)
            self.rtspSocket.send(request.encode("utf-8"))

            # Keep track of the sent request.
            self.requestSent = self.TEARDOWN
        else:
            return

        # Send the RTSP request using rtspSocket.
        print('\nRequest sent:\n' + request)
```

Listing 11: The sendRTSP request function

After sending the request, the server will send a reply message to confirm if the request is satisfied or not. The client will now use the function recvrtspReply to receive the reply from the server and then we will use the parseRtsp to analyze the server reply and set the suitable state.

```
    def recvRtspReply(self):
        """Receive RTSP reply from the server."""
        while True: # Constantly receive the packets
            reply = self.rtspSocket.recv(1024) #Receive the reply through the
    socket.

            if reply:
                self.parseRtspReply(reply.decode("utf-8"))

            # Stop receiving packets when theres TEARDOWN request
            if self.requestSent == self.TEARDOWN:
                break

    def parseRtspReply(self, data):
        """Parse the RTSP reply from the server.""" # analyzing the Server reply.
        #-------------
        # TO COMPLETE
        #-------------
        print("-"*40 + "\nReply received:\n" + data)
        lines = data.split('\n')
        seqNum = int(lines[1].split(' ')[1]) # Get the sequence number.

        # 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])
            # Assign the session ID
            self.sessionId = session

            # Process only if the session ID is the same
            if self.sessionId == session:
                if int(lines[0].split(' ')[1]) == 200:
                    if self.requestSent == self.SETUP:
                        # Set state to ready
                        self.state = self.READY
                        # Open RTP port.
                        self.openRtpPort()
                    elif self.requestSent == self.PLAY:
                        # Set state
                        self.state = self.PLAYING

                    elif self.requestSent == self.PAUSE:
                        # Set state
                        self.state = self.READY

                        # The play thread exits. A new thread is created on resume.
                        # self.playEvent.set()

                    elif self.requestSent == self.TEARDOWN:
                        # Set state
                        self.state = self.INIT
                        self.teardownAcked = 1

                        # Shut down and close the socket
                        self.rtspSocket.shutdown(socket.SHUT_RDWR)
                        self.rtspSocket.close()
```

Listing 12: The functions to analyze the reply and set the following states

Next, when the user clicks to 1 of the 4 options buttons on the GUI window , the respective request will be sent.

```
 def setupMovie(self):
   """Setup button handler."""
   if self.state == self.INIT:
      self.sendRtspRequest(self.SETUP)

def exitClient(self):
   """Teardown button handler."""
   self.sendRtspRequest(self.TEARDOWN)
   self.master.destroy() # Close the gui window
   print("the seq number is : " + str(self.rtspSeq) )
   if self.rtspSeq > 2:
      os.remove(CACHE_FILE_NAME + str(self.sessionId) + CACHE_FILE_EXT) #
 Delete the image.

def pauseMovie(self):
   """Pause button handler."""
   if self.state == self.PLAYING:
      self.sendRtspRequest(self.PAUSE)

def playMovie(self):
   """Play button handler."""
   if self.state == self.READY:
      # Create a new thread to listen for RTP packets
      threading.Thread(target=self.listenRtp).start()
      # self.playEvent = threading.Event()
      # self.playEvent.clear()
      self.sendRtspRequest(self.PLAY)
```

Listing 13: Button handler functions

When the user clicks the SETUP button on the window, the client will send the SETUP request to the server and if it receives a confirm message from the server, it will begin to open a RTP port to connect with the server. This action is done by calling the openRtpPort function :

```
def openRtpPort(self):
    """Open RTP socket binded to a specified port."""
    # Create a new socket to receive the packets.
    self.rtpSocket.settimeout(0.5) # Set timeout for 0.5s.

    try:
        # Bind the socket to the address using the RTP port given by the
 client user
        self.rtpSocket.bind(('',self.rtpPort))
    except:
        tkMessageBox.showwarning('Unable to Bind', 'Unable to bind PORT=%d' %
 self.rtpPort)
```

Listing 14: The openRtpPort function

After the connection is setup, the PLAY request can be called to write frame to the GUI window. First, the server will constantly send the packets to the client so the first task is to receive those packets , decode the packet header and update the frame sequence number :

```
def listenRtp(self):
    """Listen for RTP packets."""
    while True: # Loop till the end of the video.
        try:
            data = self.rtpSocket.recv(20480)
            if data:
                rtpPacket = RtpPacket()
                rtpPacket.decode(data)

                tempFrameNumber = rtpPacket.seqNum()
                print("Current frame number: " + str(tempFrameNumber))

                if tempFrameNumber > self.frameNbr:
                    self.frameNbr = tempFrameNumber
                    self.updateMovie(self.writeFrame(rtpPacket.getPayload()))
        except:
            # Close the RTP socket
            if self.teardownAcked == 1:
                self.rtpSocket.close()
                break
```

When the client receives the packets, it will write that data to a jpg file and update that image to the GUI window :

```
def writeFrame(self, data):
   """Write the received frame to a temp image file. Return the image file.
 """
   temp = CACHE_FILE_NAME + str(self.sessionId) + CACHE_FILE_EXT
   file = open(temp, "wb")
   file.write(data) # write the image to the file
   file.close()
   return temp

def updateMovie(self, imageFile):
   """Update the image file as video frame in the GUI."""
   photo = ImageTk.PhotoImage(Image.open(imageFile))
   self.label.configure(image = photo, height=288)
   self.label.image = photo
```

Listing 15: Write the frame to a jpg file and update the frame to the user window

Finally, if the user sends the TEARDOWN request, the server will stop sending packets to the client, close the GUI window and remove the picture that we write the frame to.

```
def exitClient(self):
   """Teardown button handler."""
   self.sendRtspRequest(self.TEARDOWN)
   self.master.destroy() # Close the GUI window
   print("the seq number is : " + str(self.rtspSeq) )
   if self.rtspSeq > 2:
      os.remove(CACHE_FILE_NAME + str(self.sessionId) + CACHE_FILE_EXT) #
 Delete the image.
```

Listing 16: The function used to close the GUI window and stop receiving the packets

If the clients explicitly click the exit symbol on the GUI window, we will pause the video and ask the user if they really want to exit. If yes, we will call the exitClient function and if no then we continue play the video.

```
def handler(self):
   """Handler on explicitly closing the GUI window."""
   self.pauseMovie()
   if tkMessageBox.askokcancel("Warning","You want to quit now ?"):
      self.exitClient()
   else:
      self.playMovie()
```

8 Evaluation of achieved results

After having completed all the required task and TODO section in the source code, we can finally connect and start the application GUI window.

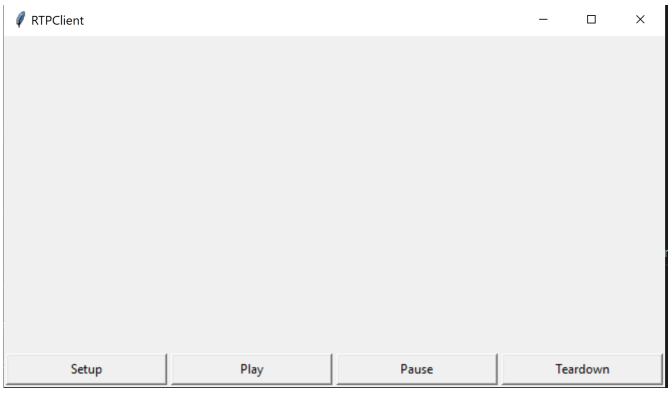
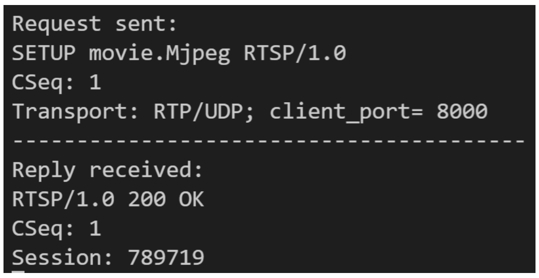


Figure 5: The GUI window

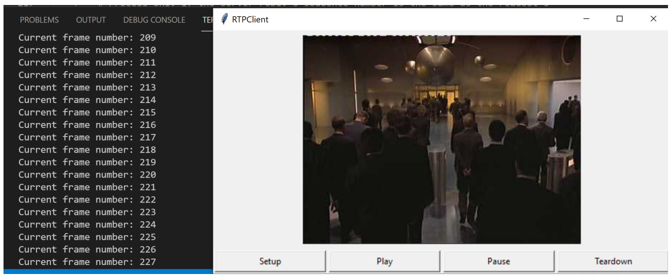
First, we will try to send the SETUP request to establish the connection by clicking the SETUP button on the GUI window and here is the result :



```
Request sent:
SETUP movie.Mjpeg RTSP/1.0
CSeq: 1
Transport: RTP/UDP; client_port= 8000
----------------------------------------
Reply received:
RTSP/1.0 200 OK
CSeq: 1
Session: 789719
```

Figure 6: The SETUP request and reply on the client side

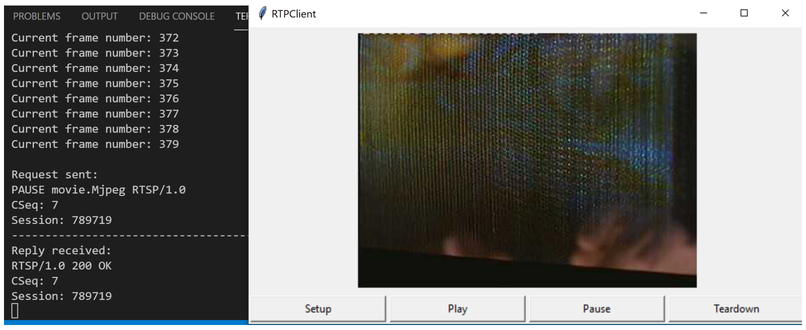
As the setup is completed, we can now play the video by clicking the PLAY button:



```
Current frame number: 209
Current frame number: 210
Current frame number: 211
Current frame number: 212
Current frame number: 213
Current frame number: 214
Current frame number: 215
Current frame number: 216
Current frame number: 217
Current frame number: 218
Current frame number: 219
Current frame number: 220
Current frame number: 221
Current frame number: 222
Current frame number: 223
Current frame number: 224
Current frame number: 225
Current frame number: 226
Current frame number: 227
```

Figure 7: Play the video

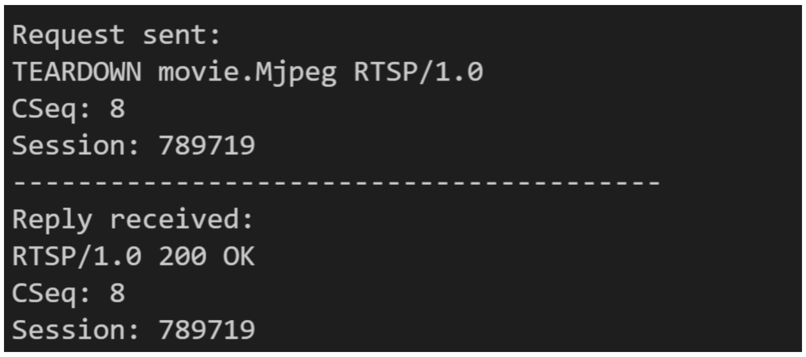
While playing the video, you can choose to pause the video when clicking the Pause button:



```
Current frame number: 372
Current frame number: 373
Current frame number: 374
Current frame number: 375
Current frame number: 376
Current frame number: 377
Current frame number: 378
Current frame number: 379

Request sent:
PAUSE movie.Mjpeg RTSP/1.0
CSeq: 7
Session: 789719
----------------------------------
Reply received:
RTSP/1.0 200 OK
CSeq: 7
Session: 789719
```

Figure 8: Pause the video

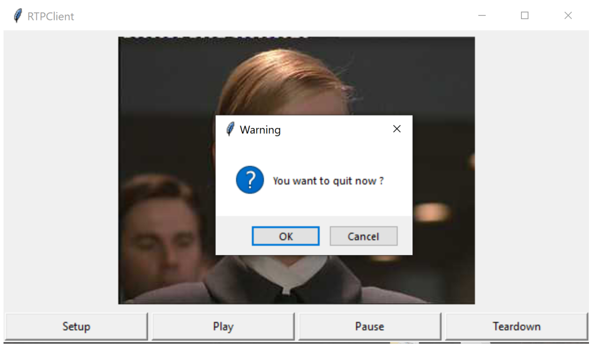
You can also choose to tear down the video :



```
Request sent:
TEARDOWN movie.Mjpeg RTSP/1.0
CSeq: 8
Session: 789719
----------------------------------------
Reply received:
RTSP/1.0 200 OK
CSeq: 8
Session: 789719
```

Figure 9: Close the video

When you click the close symbol on the GUI window, a warning will pop up. If you choose OK then the GUI window will be closed, and if you choose Cancel then the video will continue playing :



You want to quit now ?

Figure 10: Ask the user if they want to close or not

9 User manual

In order to see the final product in action, we will need to follow a few steps:

- **Step 1: Run "python Server.py <port number>"**
  - Open a command prompt or one of its equivalents on your machine.
  - Change your directory to the one containing all of the source files.
  - Make sure python is installed on your machine. Type the command :

    python Server.py <server_port>

    Where <server_port> is an argument used to bind the socket for our server. It can be any number between 1025 and 65535 (inclusive).
  - For example:

    python Server.py 1300

- **Step 2: Run "python ClientLauncher.py <server_name> <server_port> <client_port> <media_file> "**
  - Open a second command prompt or one of its equivalents on your machine.
  - Change your directory to the one containing all of the source files.
  - Make sure python is installed on your machine. Type the command :

    python ClientLauncher.py <server_name> <server_port> <client_port> <media_file>

    Where : <server_name> is the name or address of the server. In our case, we will be running this on the same machine so this can be the name of your machine or your local IP address. <server_port> is the port of our server. This must be the same as the number we provided in Step 1. The <client_port> is the port used to bind the socket of our client. <media_file> is the name of the MJPEG movie file that we want to stream or watch.
  - For example:

    python ClientLauncher.py DESKTOP-CPBGK5S 1300 5008 movie.Mjpeg

- **(Optional) Step 2.5: Fast launch on Windows OS**
  - If you wish to avoid the hassle of having to go through both Step 1 and 2, you can create a batch script file that will automatically do that for you. The whole system will be accessible with a single double-click like most executables.
  - Navigate to the directory containing the source files. Create a new Text Document by right-clicking on an empty space. This will be our batch script.
  - Open the text file and enter the following lines:

```
@echo off
start python <server_port>
start python ClientLauncher.py <server_name> <server_port> <
    client_port> <media_file>
```

- The arguments will be the same as if you were to run these commands manually. Save the text file and change its extension from ".txt" to ".bat".
- Double-click our new batch script.

- **Step 3: Interact with the client's GUI**
  - Press the SETUP button to connect our client to the server.
  - Press the PLAY button to signal the server to start sending each frame of the movie to the client. This will only function if our connection to the server exists.
  - Press the PAUSE button to signal the server to stop sending each frame of the movie to the client. This will only function if our connection to the server exists.
  - Press the TEARDOWN button to terminate our connection to the server. This will close the client.
  - Should you wish to close the client application like any regular application, there is always the close button in the top right corner. Pressing this button will prompt the user to confirm their action of closing the application. The movie will be paused during this process. If the user chooses not to close the application, the movie will begin playing again.

10 Extended features

10.1 Extension 1 : Basic statistic

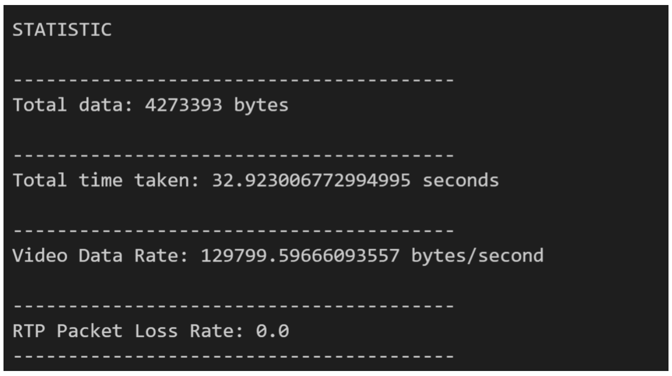
The basic idea is quite simple. In order to calculate the video data rate, you will have to calculate the total amount of transferred data (in bytes) and the total amount of time it took to transfer all the frames. Then, we just simply take the division of the total amount of data and the transfer time. In order to calculate the amount of data that has been transferred we can simply add all the length (in Bytes) of all the frames from the client. And to calculate the time, we will store the time when the first packet arrives as the begin time and when there are no packets left denoted as the end time we can take the subtraction between the end time and the begin time to find the transfer time. Then what is left is the job for simple math.

For the packet loss rate, we will take the division of the number of loss packets and the total packets sent by the server. When a packet arrives at the client, the frame number of that packet should be larger than the current frame number assigned for the client (self.frameNbr) by **1**. Else, it is considered as the late packet. By that way, we can calculate the number of loss packets. We can easily get the number of frames that have been transfer by calling the self.frameNbr when the application has been torn down.
We will make a statistic about 4 values : The video transmission rate (Byte/s), The total number of bytes (Bytes), the transfer time and the packet loss rate.

```
    # STATISTIC SECTION
    if self.sumOfTime > 0:
        print("\n STATISTIC \n")
        print('-'*40 + "\nTotal data: " + str(self.sumOfData) + " bytes \n")
        print('-'*40 + "\nTotal time taken: " + str(self.sumOfTime) +" seconds
\n")
        rateData = float(int(self.sumOfData)/int(self.sumOfTime))
        print('-'*40 + "\nVideo Data Rate: " + str(rateData) +" bytes/second\n
")
        rateLoss = float(self.countPacket/self.frameNbr)
        print('-'*40 + "\nRTP Packet Loss Rate: " + str(rateLoss))
        print('-'*40)
```

Listing 17: The statistic section

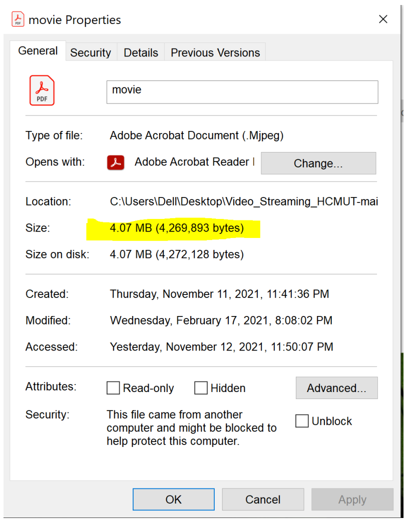
Here is the result:



```
STATISTIC

---------------------------------------
Total data: 4273393 bytes

---------------------------------------
Total time taken: 32.923006772994995 seconds

---------------------------------------
Video Data Rate: 129799.59666093557 bytes/second

---------------------------------------
RTP Packet Loss Rate: 0.0
---------------------------------------
```

Figure 11: The result of the statistic

You can see that we have a total of 4273393 bytes which equals 4.07 MB and it is the same as the size of the movie.Mjpeg file :



	movie
Type of file:	Adobe Acrobat Document (.Mjpeg)
Opens with:	Adobe Acrobat Reader I
Location:	C:\Users\Dell\Desktop\Video_Streaming_HCMUT-mai
Size:	4.07 MB (4,269,893 bytes)
Size on disk:	4.07 MB (4,272,128 bytes)
Created:	Thursday, November 11, 2021, 11:41:36 PM
Modified:	Wednesday, February 17, 2021, 8:08:02 PM
Accessed:	Yesterday, November 12, 2021, 11:50:07 PM
Attributes:	☐ Read-only ☐ Hidden
Security:	This file came from another computer and might be blocked to help protect this computer. ☐ Unblock

Figure 12: The properties of the movie.Mjpeg

You can view the full source code for this extension in the MyExtension1 folder.

10.2 Extension 2 : Media player

Based on the requirement, a standard media player such as RealPlayer or Windows Media Player, we can see that they have only 3 buttons for the same actions: PLAY, PAUSE, and STOP (roughly corresponding to TEARDOWN). There is no SETUP button available to the user. Therefore, to tackle this problem we include the function of SETUP to the function of PLAY. In order to do that, we will first remove the SETUP button on the GUI window.

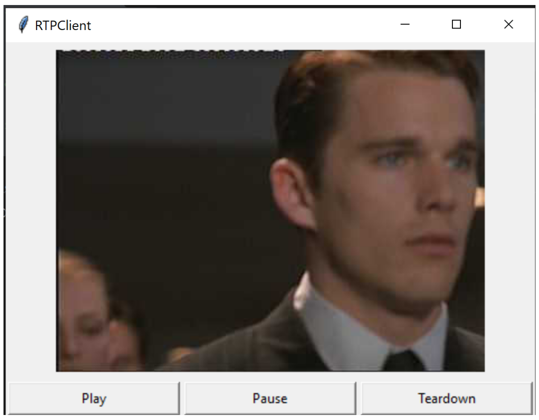


Figure 13: The new GUI window

And that is for the user interface part, for the coding part, our idea is to send the SETUP request when the user clicks to the PLAY button to establish the connection first then we will set the state to READY then open the RTP port to execute the PLAY request at the same time. This will be implemented in the playMovie function in the Client.py.

```
def playMovie(self):
  """Play button handler."""
  if self.state == self.INIT:
     self.sendRtspRequest(self.SETUP)
     self.state = self.READY
     # Open RTP port.
     self.openRtpPort()
  if self.state == self.READY:
     # Create a new thread to listen for RTP packets
     threading.Thread(target=self.listenRtp).start()
     self.sendRtspRequest(self.PLAY)
```

Listing 18: The implementation of SETUP and PLAY at the same time

You can view the full source code for this extension in the MyExtension2 folder.

10.3 Extension 3 : Add Description function

Currently, the client and server only implement the minimum necessary RTSP interactions and PAUSE. We will implement the method DESCRIBE which is used to pass information about the media stream. When the server receives a DESCRIBE request, it sends back a session description file which tells the client what kinds of streams are in the session and what encodings are used. In order to implement this task, we will add another button called Describe into the GUI window :

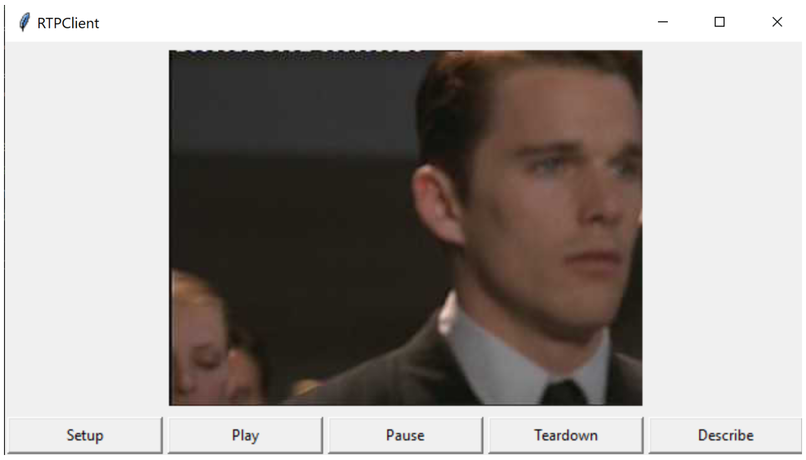


Figure 14: The GUI window after adding the Describe button.

In the Client.py file we we will add another section to sendRtspRequest function to send the DESCRIBE request from the client :

```
    #Describe request
    elif requestCode == self.DESCRIBE:
        self.rtspSeq = self.rtspSeq + 1
        request = "DESCRIBE " + str(self.fileName) + " RTSP/1.0\nCSeq: " + str
  (self.rtspSeq) + "\nSesssion: " + str(self.sessionId)
        self.rtspSocket.send(request.encode("utf-8"))
```

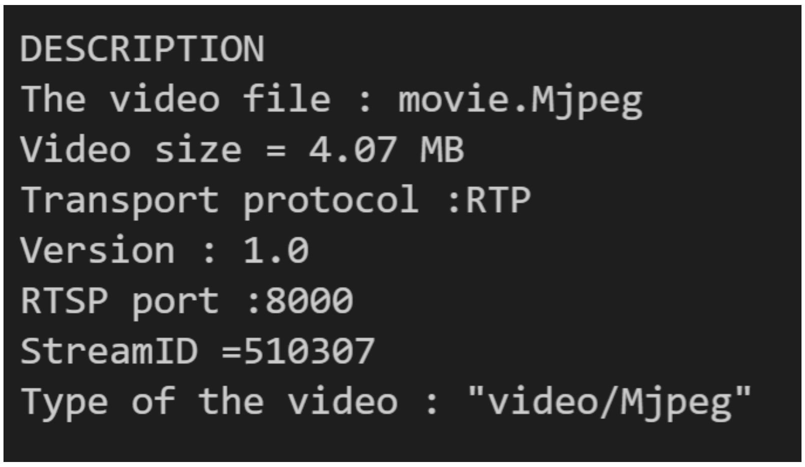
Listing 19: Send DESCRIBE request to the server.

For the Server.py, we add will add a function to return the description about the streaming protocol, the version of RTSP, the video name that is played, the length of that video, the RTSP port and the streaming session ID. This description is transferred along with a OK reply :

```
  def RepDescribe(self,code,seq):
    # Basic description
    description = description = "\nDESCRIPTION\nThe video file : " + str(self
  .clientInfo['videoStream'].filename) + "\nVideo size = " \
    + str(round(os.path.getsize(self.clientInfo['videoStream'].filename)
  /(1024*1024),2)) + " MB" + "\nTransport protocol :"\
    "RTSP\nVersion : 1.0\nRTP port :"+str(self.clientInfo['rtpPort'])+ "\
  nStreamID =" \
    + str(self.clientInfo['session']) +"\nType of the video : \"video/Mjpeg
\"\n"

    if code == self.OK_200:
        reply = "RTSP/1.0 200 OK\nCSeq: " + seq + "\nSession: " + str(self.
clientInfo['session']) + "\n" + description
        connSocket = self.clientInfo['rtspSocket'][0]
        connSocket.send(reply.encode())
    elif code == self.FILE_NOT_FOUND_404:
        print("404 FILE NOT FOUND")
    elif code == self.CON_ERR_500:
        print("500 CONNECTION ERROR")
```

Listing 20: Reply to the DESCRIBE request and send the description to the client.

And this is the sample description script which is sent by the server when the client send the Describe request :



```
DESCRIPTION
The video file : movie.Mjpeg
Video size = 4.07 MB
Transport protocol :RTP
Version : 1.0
RTSP port :8000
StreamID =510307
Type of the video : "video/Mjpeg"
```

Figure 15: The description

10.4 Extension 4: Backward, forward and restart

The main idea of extension 4 will be about upgrading the client's GUI. Primarily, we will implement 4 extra features in the client, including 3 buttons and 1 label. The 3 buttons will be about restarting the video, go back 20 frames or skip ahead 20 frames. The label will be showing us the current frame number over the total number of frames in the movie being streamed.

First, we will need to "upgrade" our RTSP version to be able to handle the new features. We will be naming our new version of the RTSP, version 1.4, which means the header of all RTSP messages will now include RTSP/1.4 instead of RTSP/1.0. The big difference between these 2 versions is that, RTSP/1.4 can process 3 more requests and include 1 more header field. Those are the RESTART, BACKWARD and FORWARD requests and the Frames header field.

Starting with Client.py, we will add a few tkinker elements to our widget to represent the new GUI. We will also add functions that are binded to those new elements, whose main goal is to send RTSP request to the server with their respective request type. When the client receives a good reply from the server (with status 200 OK), depending on our sent request, certain attributes will be updated such as decreasing the frame number, updating the label to represent time...

The revamped user interface now looks like this:

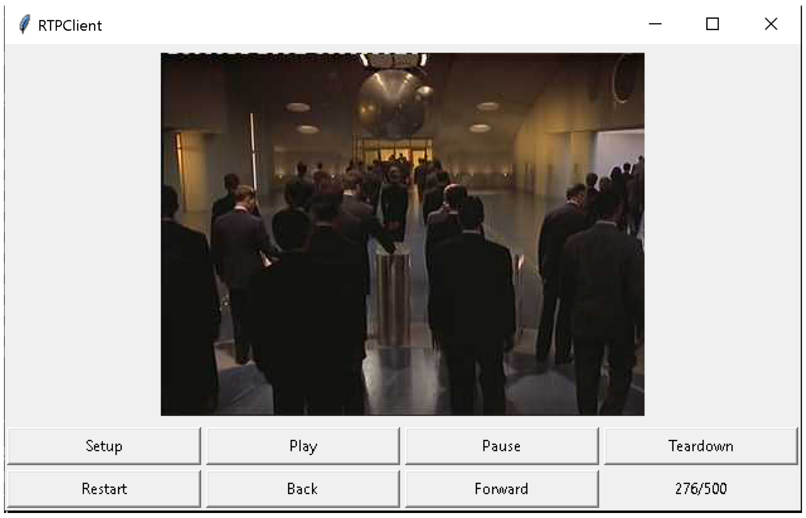


Figure 16: The new GUI window

Moving onto the middleman, ServerWorker.py. We will add the handle for the 3 new requests and the actions that come with receiving those. The RTSP reply sent by the server will also include the new header field Frames, which carry the total number of frames in our movie. The ServerWorker.py won't be doing much regarding the new requests. Rather, its job is to signal the attached VideoStream object to adjust itself as that is where the majority of the information pertaining to our movie resides.

VideoStream.py is perhaps the file that received the biggest number of changes. In order to wind a video backwards, we need to know how much the previous frame is in terms of length. As there is no way of knowing that from any given frame, we have decided to include a rather lengthy list that accounts for the position of every frame in the movie. Crucially, we have added a probe function to gather every piece of information we need. Its main job is to run through the whole movie once when the SETUP button is pressed. Every time it passes by a frame, it remembers the position of that frame with respect to the file and stores it in our list and it also increases the total frame count by 1. At the end of the function, we will have obtained the total number of frames in the movie and the position needed to jump directly to any frame we desire. The total number of frames will be returned to ServerWorker.py to be included in the header. As for the "jump", it is performed mainly with the help of the seek function provided in the file API of python, moving the pointer within the file we're reading from to the position we want it to be in the form of the argument provided, being the position we saved from earlier. The list allows us to maintain an access time of O(1) regardless of how many frames there are as long as we can pinpoint the exact frame that we need. This is a huge payout for the price of using up the extra space for our list and the reason why we agreed on going through with this decision as low access time is important in a real-time streaming service. And finally, we have also added a few smaller functions to help with the trivial task of managing the frame number when there's a RESTART, BACKWARD or FORWARD request, and consequentially performing the jump to said frame number.

In our implementation, we have decided that a FORWARD and BACKWARD request should move the video by 20 frames as it is the ideal number of frames sent per second (the interval is 50 milliseconds). However the actual number of frames showed in 1 second does not match up with the calculated number, and it varies greatly between each of our member's machine and thus we cannot determine what the exact number of frames we should skip is. It is also the reason we have chosen to let the time label aspect of our client use frames as the time unit instead of seconds.

```python
    def probe(self, fn):
      """Run through whole file first to get info"""
      file1 = open(fn, 'rb')
      currentPos = 0
      # Get pos of every frame and also count total frames
      while file1.read(5):
         # Move pointer back to compensate for condition check
         file1.seek(file1.tell() - 5)
         # Advance 1 frame
         frameLength = int(file1.read(5))
         file1.read(frameLength)
         currentPos += frameLength + 5
         # Push pos into array
         self.posArray.append(currentPos)
         # Increase count
         self.totalFrame += 1

      # Pop the last cell since it's just pos of eof
      print(self.posArray.pop())
```

```
        file1.close()

    def toFrame(self, frameNum):
        """Move file to pos of frame number frameNum"""
        self.file.seek(self.posArray[frameNum])
```

Listing 21: The core functions of this extension, probe and toFrame

Note : You can view the full source code for this extension in the MyExtend4 folder.

11 Conclusion

By finishing this assignment, we have understood more about Real-Time Streaming Protocol (RTSP) as well as how it works and its applications in building Video streaming application. We have also completed a simple Video streaming application based on RTSP to show what we have learned so far about RTSP. Furthermore, we have also included some extended features like adding basic statistics to verify the integrity of the app, merge the set up request to the play request, add a description request and add the forward, backward, restart features to the video application and done our researches about port forwarding. The process has proven itself to be a challenging and daunting task but we believe this assignment has given us plenty of knowledge and experience needed for developing network applications in the future.

12 Source code

We already include the source code in the submission but here is the git hub link to our source code :
https://github.com/huycao2001/Computer-Network-Lab-Video-streaming-application.git