VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Laboratory

COMPUTER NETWORKS ASSIGNMENT 1

Lecturer: Assoc. Prof. Tran Van Hoai.

TA: Mr. Nguyen Manh Thin.
Students: Tran Hung Cuong - 1952606.
Nguyen Vo Hoang Thi - 1952996.
Nguyen Truc Phuong - 1952402.

HO CHI MINH CITY, 2021

Contents

1 Requirements

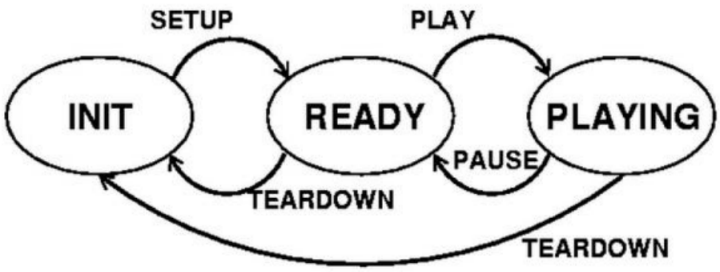
1.1 Functional Requirements

A streaming video server communicate with the client by using the Real-Time Streaming Protocol (RTSP) and send data using the Real-time Transfer Protocol (RTP).

- Server:
  1. Server is able to perform communication with client via RTSP/RTP protocol.
  2. Server can respond to RTSP request and stream video.
- Client:
  1. It is possible for client to connect to the server through the terminal.
  2. User can perform task via 4 main button: Setup, Play, Pause and Teardown.
  3. The server replies to all messages that the client sends.
  4. Create a datagram socket for receiving RTP data.

1.2 Non-Functional Requirements

- The port number is greater than 1024.
- There are 3 reply codes:
  1. 202: the request is successful.
  2. 404: FILE_NOT_FOUND error.
  3. 505: connection error.
- The timeout on the datagram socket which receive RTP data is 0.5 seconds.
- Every requests that are sent by client should have CSeq header value attached with it.
- Keep the client's state up to date. Client changes state when it receives a reply from the server according to the following state diagram:



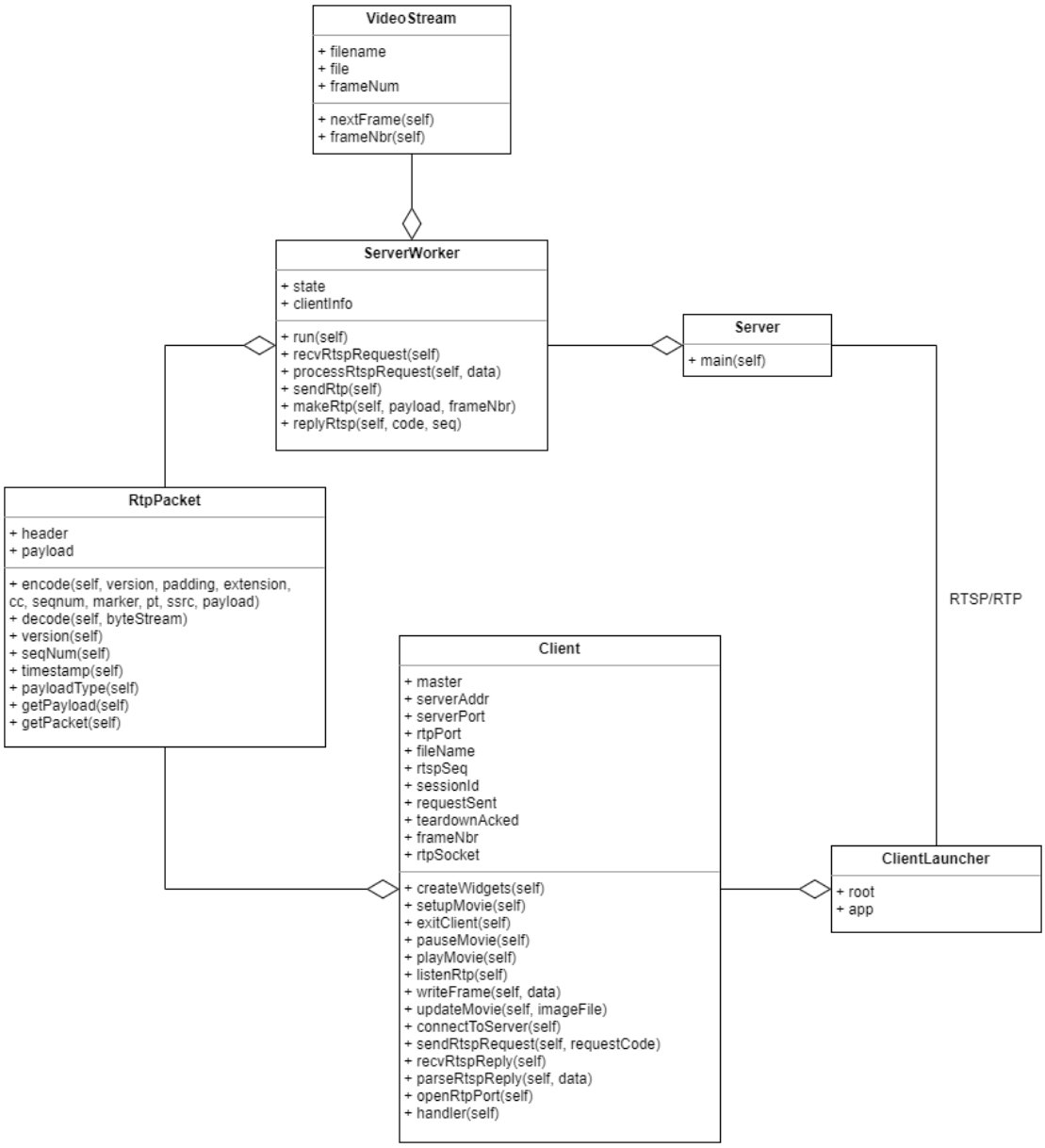
- The frame is sent to the client over UDP every 50 milliseconds.
- The format of the video is .Mjpeg (Motion JPEG).

2 Functions Descriptions

Function	Parameter	Description
__init__(sefl, clientInfo)	sefl, clientInfo	Constructor
run(self)	self	Run the server
processRtspRequest(self, data)	self, data	Process RTSP request
sendRtp(self)	self	Send RTP packets over UDP
makeRtp(self, payload, frameNbr)	self, payload, frameNbr	RTP-packetize the video data
replyRtsp(self, code, seq)	self, code, seq	Send RTSP reply to client
main(self)	self	Main function executing the whole program
__init__(self, filename)	self, filename	Constructor
nextFrame(self)	self	Get the next frame
frameNbr(self)	self	Get the frame number
__init__(self, master, serveraddr, serverport, rtpport, filename)	self, master, serveraddr, serverport, rtpport, filename	Constructor
createWidgets(self)	self	Create the GUI
setupMovie(self)	self	Setup button handler
exitClient(self)	self	Teardown button handler
pauseMovie(self)	self	Pause button handler
playMovie(self)	self	Play button handler
listenRtp(self)	self	Listen for RTP packets
writeFrame(self, data)	self, data	Write the received frame to a temp image file. Return the image file
updateMovie(self, imageFile)	self, imageFile	Update the image file as video frame in the GUI
connectToServer(self)	self	Connect to the Server. Start a new RTSP/TCP session
sendRtspRequest(self, requestCode)	self, requestCode	Send RTSP request to the server
recvRtspReply(self)	self	Receive RTSP reply from the server
parseRtspReply(self, data)	self, data	Parse the RTSP reply from the server
openRtpPort(self)	self	Open RTP socket binded to a specified port

Function	Parameter	Description
handler(self)	self	Handler on explicitly closing the GUI window
__init__(self)	self	Constructor
encode(self, version, padding, extension, cc, seqnum, marker, pt, ssrc, payload)	self, version, padding, extension, cc, seqnum, marker, pt, ssrc, payload	Encode the RTP packet with header fields and payload
decode(self, byteStream)	self, byteStream	Decode the RTP packet
version(self)	self	Return RTP version
seqNum(self)	self	Return sequence (frame) number
timestamp(self)	self	Return timestamp
payloadType(self)	self	Return payload type
getPayload(self)	self	Return payload
getPacket(self)	self	Return RTP packet

3 Class Diagram



4 Implementation

When the client starts, it also opens the RTSP socket to the server. This socket will be used to send all RTSP requests. Client is provided 4 buttons according to 4 action that client can interact with the server.

- SETUP
- PLAY
- PAUSE
- TEARDOWN

These actions will send RTSP requests to the server and tell the server which tasks to be archived. And after receiving commands, the server will send the messages, through RTSP protocol, to the client to inform whether the requests are successful or not. There are 3 types of messages which are already discussed in the requirements section.

4.1 SETUP

When client click the SETUP button, a datagram socket fro receiving RTP data will be created and a SETUP request will be sent. When receiving the request, the server performs the following tasks:

- Check whether the video file is available or not, set the state to READY. If video file is available, the server will set the frame number into 0, or else the server will send a 404 FILE_NOT_FOUND messages.
- Create a random ID from 100000 - 999999.
- Return a message for client.

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

```python
  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) + " 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.requestSent = self.SETUP
```

4.2 PLAY

When client click the PLAY button, the function *playMovie()* will be executed:

```
    def playMovie(self):
      """Play button handler."""
      if self.state == self.READY:
        self.trigle = True
        # 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)
```

A thread will be created to listen for RTP packets and send a PLAY request to the server.

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

      # 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 = ...
        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.requestSent = self.PLAY
```

When the server receive the request, it will execute the request:

```python
def recvRtspRequest(self):
  """Receive RTSP request from the client."""
  connSocket = self.clientInfo['rtspSocket'][0]
  while True:
    data = connSocket.recv(256)
    if data:
      print("Data received:\n" + data.decode("utf-8"))
      self.processRtspRequest(data.decode("utf-8"))
```

Server will create a socket to send RTP via UDP and start to send packet video. Before sending packet, server will send a message to the client to inform. The frame will be sent to the client over UDP every 50 miliseconds. A function which format the packet alse is provided.
For the encapsulation, the server calls the encode function of the RtpPacket class.

```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())
    header = bytearray(HEADER_SIZE)
    self.header = bytearray(HEADER_SIZE)
    #--------------
    # TO COMPLETE
    #--------------
    # Fill the header bytearray with RTP header fields

    # header[0] = ...
    # ...
    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
    # Get the payload from the argument
    # self.payload = ...
```

```
    self.payload = payload
```

Client will receive and decode the RTP packet with the fuction *listenRTP()* and *decode()*. And function *updateMovie()* will update the image file as a video frame in the GUI.

4.3 PAUSE

When the state of client is PLAYING and there is a PAUSE request sent by client, server will temporary stop sending packet to client by setting PAUSE requestSent.

```
elif requestCode == self.PAUSE and self.state == self.PLAYING:
      # Update RTSP sequence number.
      # ...
      self.rtspSeq = self.rtspSeq + 1

      # Write the RTSP request to be sent.
      # request = ...
      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.requestSent = self.PAUSE
```

When the server receive PAUSE request from the client, the state will be changed into READY and the server will wait for next client's 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])
```

4.4 TEARDOWN

As discussed before, this request will terminate the session and close the connection. Client's state will be changed into INIT, and the server will close the RTP socket.

```
def exitClient(self):
    """Teardown button handler."""
    self.sendRtspRequest(self.TEARDOWN)
    self.master.destroy() # Close the gui window
    os.remove(CACHE_FILE_NAME + str(self.sessionId) + CACHE_FILE_EXT) #
    Delete the cache image from video
```

```
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()
```

5 A summative evaluation of achieved results

After complete the code file, we archived following results:

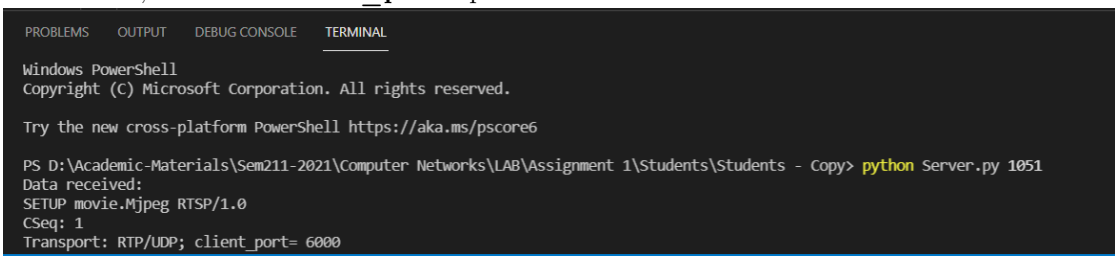
- New GUI for client.
- Complete the RTDP protocol.
- Complete the RTP Packetization in the server.
- Messages request and reply are sent by client and server.

6 User Manual

Step 1: We run the terminal and start the server with the command:

python Server.py server_port

Server_port is the port that server listens to for incoming RTSP connections.The standard RTSP port is 554, but we will choose a port number greater than 1024.
In our work, we choose **server_port** equal 1051.



Step 2: Open a new terminal and start the client with the command:

python ClientLauncher.py server_host server_port RTP_port video_file

Where:

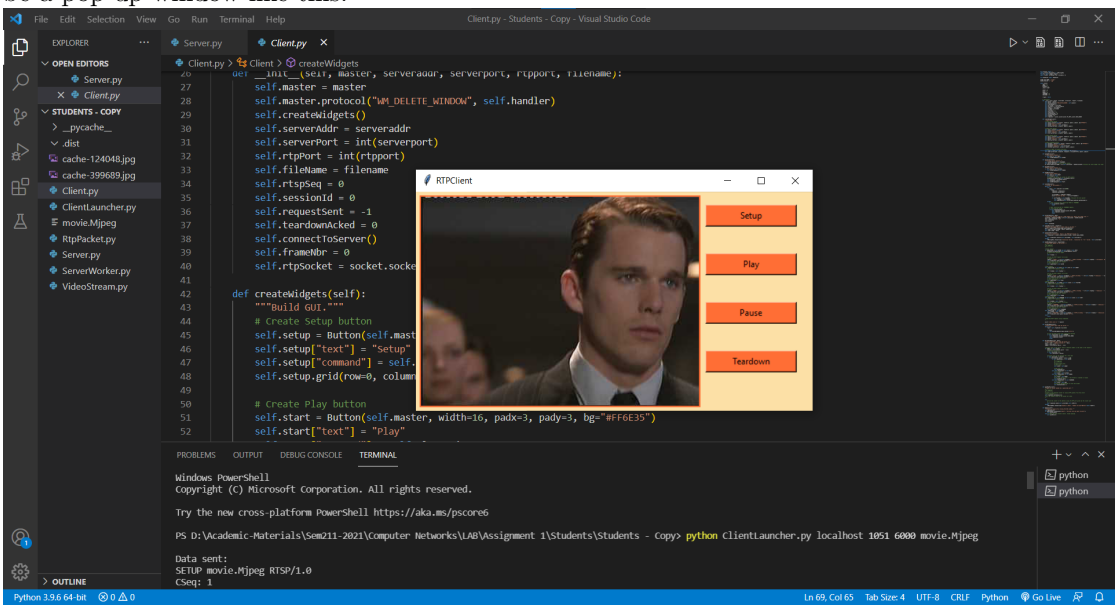
- **server_host:** the name of the machine where the server is running.
- **server_port:** the port where the server is listening on.
- **RTP_port:** the port where the RTP packets are received.

- **video_file:** the name of the video (we have been provided an example file named movie.Mjpeg).

We choose:

- **server_host:** localhost.
- **server_port:** 1051 (created in the first step).
- **RTP_port:** 6000.
- **video_file:** movie.Mjpeg.

After running these above command. The server create a connection to the client and there will be a pop-up window like this:



Step 3: Now, the client can send RTSP request to the server via 4 buttons on the GUI:

1. **SETUP:** create the RTP connection to the server.
2. **PLAY:** this request will ask the server to start playing the video.
3. **PAUSE:** pause the video.
4. **TEARDOWN:** close the connection to the server, also close the GUI.

7 Extend

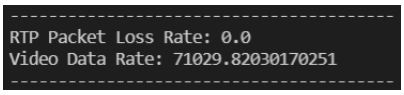
7.1 Question 1

Calculate the statistics about the session. You will need to calculate RTP packet loss rate, video data rate (in bits or bytes per second), and any other interesting statistics that you can think of.

Here is the calculation for RTP packet loss rate and Video data rate:

```python
    def exitClient(self):
        """Teardown button handler."""
        self.sendRtspRequest(self.TEARDOWN)
        self.master.destroy() # Close the gui window
        rateloss = float(self.packetLoss / self.frameNbr)
        datarate = float(float(self.totalData)/float(self.totalTime))
        print('-'*40 + "\nRTP Packet Loss Rate: " + str(rateloss))
        print("Video Data Rate: " + str(datarate) +"\n" + '-'*40)
        os.remove(CACHE_FILE_NAME + str(self.sessionId) + CACHE_FILE_EXT) #
        Delete the cache image from video
```

After terminating the session, the RTP packet loss rate and Video data rate will be displayed on the screen:



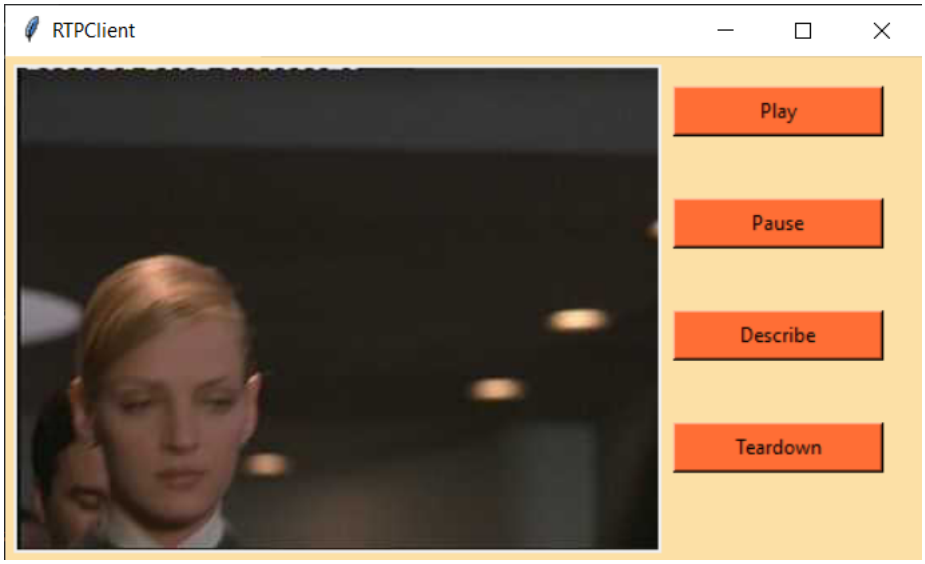
7.2 Question 2

The user interface on the RTPClient has 4 buttons for the 4 actions. If you compare this to a standard media player, such as RealPlayer or Windows Media Player, you 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. Given that SETUP is mandatory in an RTSP interaction, how would you implement that in a media player? When does the client send the SETUP? Come up with a solution and implement it. Also, is it appropriate to send TEARDOWN when the user clicks on the STOP button?

Here, we remove the SETUP button, when ClientLauncher start running, SETUP request will be sent concurrently. So when we start streaming the video, the state of client is READY.

```python
    def __init__(self, master, serveraddr, serverport, rtpport, filename):
        self.master = master
        self.master.protocol("WM_DELETE_WINDOW", self.handler)
        self.createWidgets()
        self.serverAddr = serveraddr
        self.serverPort = int(serverport)
        self.rtpPort = int(rtpport)
        self.fileName = filename
        self.rtspSeq = 0
        self.sessionId = 0
        self.requestSent = -1
        self.teardownAcked = 0
        self.connectToServer()
        self.frameNbr = 0
        self.rtpSocket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
        self.setupMovie()
```

And after that, client just use 3 buttons: PLAY, PAUSE, TEARDOWN to interact with the server:



7.3 Question 3

Currently, the client and server only implement the minimum necessary RTSP interactions and PAUSE. 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 this section, our group add 1 more button: DESCRIBE:

```
  def describe(self):
    if self.trigle:
      self.sendRtspRequest(self.DESCRIBE)
```

```
elif requestCode == self.DESCRIBE:
      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"))
```

And here is what the server will do when it receive a DESCRIBE request:

```
      # Take server port from str
    str1 = str(self.clientInfo['rtspSocket'][0])
    server_port = ''
    count = 0
    count1 = 0
    for i in range(len(str1)):
      if str1[i] == '(':
        count1 += 1
```

```
        if str1[i] == ',' and count1 == 1:
          count += 1
        if count == 1 and str1[i] != ',' and str1[i] != ' ':
          if str1[i] == ')':
            break
          server_port += str1[i]

      seq1 = "Client port: " + str(self.clientInfo['rtpPort']) + " RTP/AVP 26\n" + "Server port: " + server_port + "\na=control:streamid=" \
         + str(self.clientInfo['session'])
         +"\na=mimetype:string;\"video/Mjpeg\"\n"+ '-'*40
      seq2 = 'DESCRIBE:\n' + '-'*40 + "\nContent-Base: " + str(self.clientInfo['videoStream'].filename) +"\n"
      return seq2 + seq1
```

```
    def replyDescibe(self,code,seq):
      des = self.describe()
      if code == self.OK_200:
        reply = "RTSP/1.0 200 OK\nCSeq: " + seq + "\nSession: " + str(self.clientInfo['session']) + "\n" + des
        connSocket = self.clientInfo['rtspSocket'][0]
        connSocket.send(reply.encode())

      # Error messages
      elif code == self.FILE_NOT_FOUND_404:
        print("404 NOT FOUND")
      elif code == self.CON_ERR_500:
        print("500 CONNECTION ERROR")
```

And here is what will be displayed:

```
DESCRIBE:
----------------------------------------
Content-Base: movie.Mjpeg
Client port: 6000 RTP/AVP 26
Server port: 1051
a=control:streamid=103162
a=mimetype:string;"video/Mjpeg"
----------------------------------------
```

The information includes:

- Content-Base: the name of the video.
- Client port.
- Server port.
- a=control: ID of the session.
- a=mimetype: type of the video.

7.4 Question 4

Implement some additional functions for user interface such as: display video total time and remaining time, fast forward or backward video (or make a scroll bar for scrolling video if you can).

7.5 Question 5

Add one more state to the client (for example SWITCH state) so that user can select another video from a list of videos received from server.