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

COMPUTER NETWORK

Assigment 1

REAL-TIME STREAMING PROTOCOL AND TRANSFER PROTOCOL

GVHD: Bùi Xuân Giang
SV: Huỳnh Thị Uyên - 1810648
Võ Hoàng Hải Nam - 1810340
Lê Thành Lâm - 1810730

HO CHI MINH CITY, 11/2020

Mục lục

Danh sách hình vẽ

Danh sách bảng

Work assignment

Student	Tasks
Huynh Thi Uyen	Introduction, Overview, Implementation
Vo Hoang Hai Nam	Class Diagram, Requirements Analysis
Le Thanh Lam	Demonstation, Evaluation, Extend, Summary

1 Introduction

In modern Internet, streaming video become more and more popular. We can see its application everywhere, especially on social networks like Facebook, Zalo or online sales pages. The application layer protocols that we often see work in a request/response manner, whereby the client asks for some piece of content, the content is delivered using TCP or UDP, and then the client application can display the content to the use. But if the user needs to watch an hour video or a movie with hundreds of megabytes of capacity, HTTP or FTP is no longer suitable for download and play.
RTSP is a solution that combines TCP to control and UDP for transport. Using this protocol, file delivery can start and the client-side application can begin displaying the audio and video content before the complete file has arrived.
In this assignment, we will use RTSP to create a simple small application to examine basicly how this procol works.

2 Overview

2.1 Real-Time Streaming Protocol

The Real Time Streaming Protocol (RTSP) is an application-level protocol that enables control over the delivery of data. It is a network control protocol designed for use in entertainment and communications systems to control streaming media servers. The transmission of streaming data is not a task of RTSP but it can use the RTP in conjunction with RTCP.
Like HTTP, RTSP defines control sequences useful in controlling multimedia playback and uses TCP to maintain the connection between endpoints. While HTTP is stateless, RTSP has state; an identifier is used when needed to track concurrent sessions.
In this assignment, we will use the following four main methods:

- **SETUP:** The client asks the server to allocate resources for a stream and start a RTSP session. This must be done before a PLAY request is sent.
  The server reply usually confirms the chosen parameters, and fills in the missing parts, such as the server's chosen ports.
  Example:
  ```
  C: SETUP movie.Mjpeg RTSP/1.0
  C: CSeq: 1
  ```

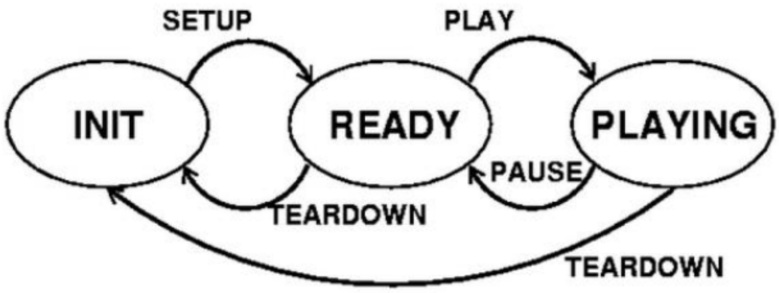
C: Transport: RTP/UDP; client_port= 25000

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

- **PLAY:** The client asks the server to start sending data on a stream allocated via SETUP.
  Example:
  C: PLAY movie.Mjpeg RTSP/1.0
  C: CSeq: 2
  C: Session: 123456

  S: RTSP/1.0 200 OK
  S: CSeq: 2
  S: Session: 123456
- **PAUSE:** The client temporarily halts the stream delivery without freeing server resources.
  Example:
  C: PAUSE movie.Mjpeg RTSP/1.0
  C: CSeq: 3
  C: Session: 123456

  S: RTSP/1.0 200 OK
  S: CSeq: 3
  S: Session: 123456
- **TEARDOWN:** The client asks the server to stop delivery of the specified stream and free the resources associated with it.
  Example:
  C: TEARDOWN movie.Mjpeg RTSP/1.0
  C: CSeq: 4
  C: Session: 123456
  S: RTSP/1.0 200 OK



Hình 1: State transition in RTSP

```
S: CSeq: 4
S: Session: 123456
```

As mentioned above, the RTSP client and server state machines describe the behavior of the protocol from RTSP session initialization through RTSP session termination. These are:

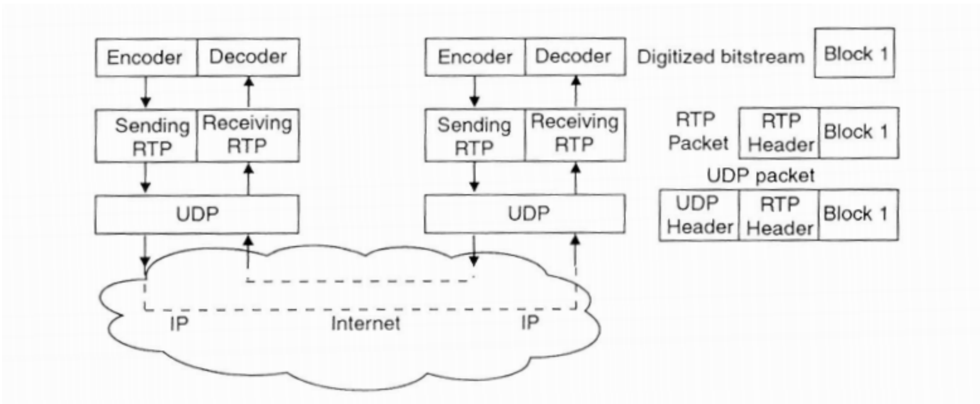
- **INIT:** SETUP has been sent, waiting for reply at client while no valid SETUP has been received yet at server.
- **READY:** SETUP reply received or PAUSE reply received while in Playing state at client and last SETUP received was successful, reply sent or after playing, last PAUSE received was successful, reply sent.
- **PLAYING:** PLAY reply received at client and last PLAY received was successful, reply sent, data is being sent.

The client and server state will be changed as messages are received. Figure 1 summarizes the client and server state changes.

2.2 Real-time Transfer Protocol

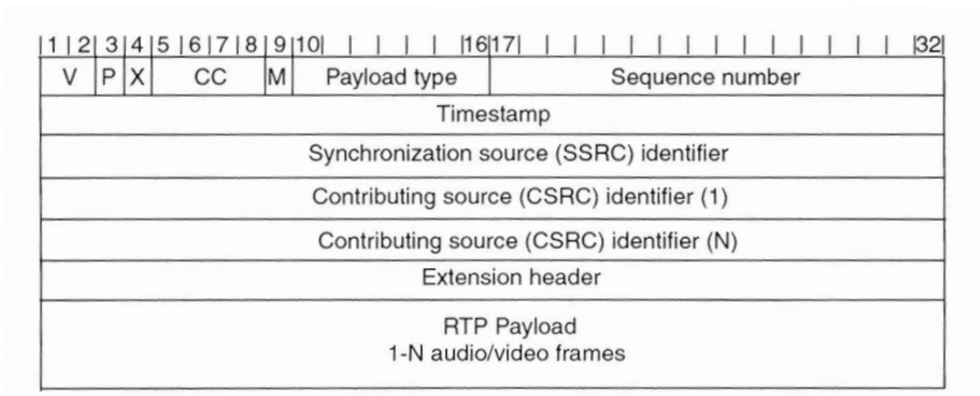
RTP is the Internet-standard protocol for the transport of real-time data, including audio and video. RTP is used in conjunction with the RTP Control Protocol (RTCP). RTP is used for the exchange of multimedia data, while RTCP is used to monitor transmission statistics and quality of service (QoS) and aids synchronization of multiple streams.

RTP typically runs on top of UDP. The sending side encapsulates a media chunk within an RTP packet, then encapsulates the packet in a UDP segment, and then hands the segment to IP. The receiving side extracts the RTP packet from the UDP segment, then extracts the media chunk from the RTP packet, and then passes the chunk to the media player for decoding and rendering.



Hình 2: Real-Time Transport Protocol

RTP packets are created at the application layer and handed to the transport layer for delivery. Each unit of RTP media data created by an application begins with the RTP packet header.



Hình 3: RTP header format

The four main RTP packet header fields are:

- The payload type (7 bits): It identifies the format of RTP payload and determines its interpretation by the application. For example, PCM, MPEG1/MPEG2 audio and video, motion JPEG, etc. Figure 4 lists some the audio and video payload types supported by RTP.
- Sequence number field (16 bits): The sequence number increments by one for each RTP packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence.
- Timestamp field (32 bits): Used to enable the receiver to play back the received samples at appropriate intervals. When several media streams are present, the timestamps are independent in each stream, and may not be relied upon for media synchronization. The granularity of the timing is application specific.
- Synchronization source identifier (SSRC): 32 bits. It identifies the source of the RTP stream. It is the number that the server creates randomly when the new stream is started.

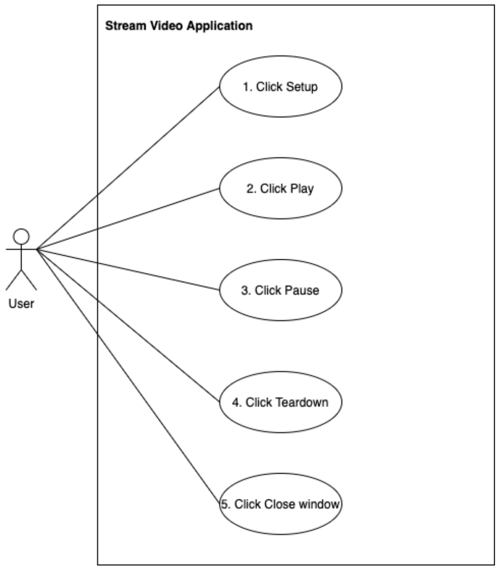
PT	Encoding name	Media type
0	PCMU	Audio
3	GSM	Audio
4	G723	Audio
5	DVI4	Audio
6	DVI4	Audio
7	LPC	Audio
8	PCMA	Audio
9	G722	Audio
10	L16	Audio
11	L16	Audio
12	QCELP	Audio
13	CN	Audio
14	MPA	Audio
15	G728	Audio
16	DVI4	Audio
17	DVI4	Audio
18	G729	Audio
25	CelB	Video
26	JPEG	Video
31	H261	Video
32	MPV	Video
33	MP2T	Audio/Video
34	H263	Video

Hình 4: Some audio and video payload types

3 Application

3.1 Requirements Analysis

3.1.1 Usecase Diagram



Hình 5: Usecase diagram

3.1.2 Functional Requirements and Function Description

Click Setup
Requirement: • Setup connection between Client App and Server.
Function description: 1. Client send SETUP request to the server 2. The Server process the request and reply to Client. If success, it will send session Id 3. The Client process reply. If success, it will set state READY open Rtp Port

Bảng 1: Click Setup requirement and description

Click Play
Requirement: • Client can watch video displayed on the screen in consecutive changing frames. • Client can't choose video's filename to watch. This is in application's build step with optional video's filename parameter.
Function description: 1. The Client starts listening to Rtp 2. The Client send PLAY request to the Server with session Id 3. The Server process the request and reply to Client. 4. The Server split video into frames, encode them and send to Client 5. The Client process reply by setting state to PLAY 6. The Client decode package and write new frame to screen.

Bảng 2: Click Play requirement and description

Click Pause
Requirement: • The video streaming pauses playing
Function description: 1. Client send PAUSE request to the Server with session Id 2. The Server process the request and reply to Client. 3. The Server stop sending package to Client. 4. The Client process reply by setting state to READY 5. The Client stop listening Rtp from the Server.

Bảng 3: Click Pause requirement and description

Click Teardown
Requirement: • The video streaming stops • Client app closes
Function description: 1. Client send TEARDOWN request to the Server with session Id 2. The Server process the request and reply to Client. 3. The Server close socket (quit session) 4. The Client process reply by setting state to INIT 5. The Client close socket 6. The Client app closes 7. All cache file will be removed

Bảng 4: Click Teardown requirement and description

Click Close window
Requirement: • The video streaming pauses • The Client chooses between close app and continue to watch video.
Function description: 1. Calling Click Pause function 2. If the Client chooses close app, calls Click Teardown function 3. If the Client chooses continuing to watch video, calls Click Play function.

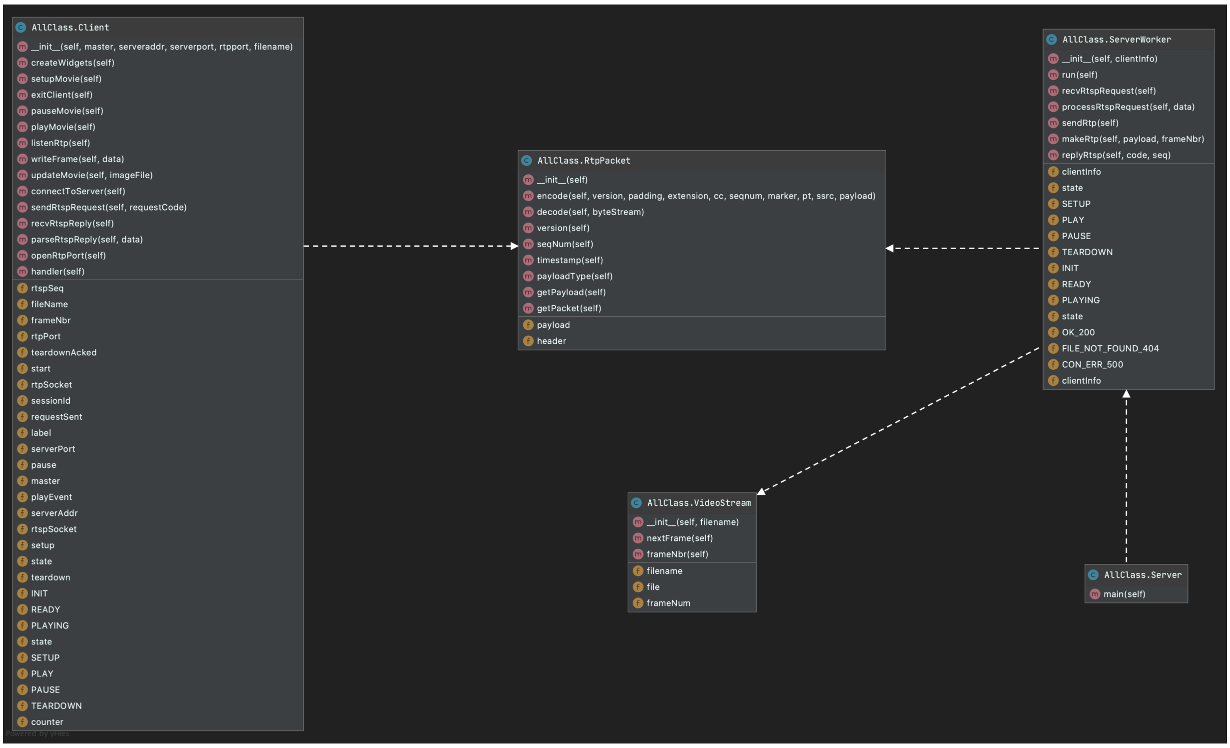
Bảng 5: Click Close window requirement and description

3.1.3 Non-functional Requirements

Non-functional Requirements:

- Support video in MJPEG (Motion JPEG) format
- Support video frame size up to 20 000 bytes
- Time for sending request and receiving the server's reponse is less than 0.05s
- Video transfer time on RTP socket is no more than 0.5s
- The rate of data loss is less than 0.1%

3.2 Class Diagram



Hình 6: Class diagram

In these classes:

1. `class Client` uses `class RtpPacket` to decode package from Server
2. `class Server` establishes connection and uses `class ServerWorker` to handle the requests.
3. `class ServerWorker` uses `VideoStream` to get video frame and `class RtpPacket` to encode and packetize the video frame.
4. `class RtpPackage` has separate methods for handling package (encode for server, decode for client,...)
5. `class VideoStream` to split video into frame and numbered packages.

3.3 Implementation

In this assignment, we will implement 4 commands: SETUP, PLAY, PAUSE, TEARDOWN. These commands will be sent from client to server via RTSP. Besides, we also have 4 states for the client and the server: SETUP, PLAY, PAUSE, TEARDOWN as mentioned in Section 2.1

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

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

The server always replies to all the messages that the client sends. This assignment has 3 codes:

- Code 200: Successful request
- Code 404: FILE_NOT_FOUND error
- Code 500: Connection error.

```
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
```

3.3.1 Client

- SETUP
  - Send `SETUP` request to the server. The `SETUP` RTSP packet includes:
    `SETUP` command, video file name, protocol type: RTSP/1.0 RTP
    RTSP Packet Sequence Number starts from 1 (Just add 1, the initial sequence number is 0)
    Transmission Protocol: UDP, RTP Port for video stream transmisison from client's input.

```
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 = 'SETUP ' + self.fileName + ' RTSP/1.0\n'+\
              'CSeq: ' + str(self.rtspSeq) + '\n'+\
              'Transport: RTP/UDP; client_port= ' + str(
 self.rtpPort)

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

  - Read the server's response and parse the Session head to get the RTSP session ID.

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

```

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

    # 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:
          #-------------
          # TO COMPLETE
          #-------------
          # Update RTSP state.
          self.state = self.READY
          # Open RTP port.
          self.openRtpPort()

```

– Create a datagram socket for receiving RTP data and set timeout on the socket to 0.5s

```
def openRtpPort(self):
  """Open RTP socket binded to a specified port."""
  #-------------
  # TO COMPLETE
  #-------------
      # Create a new datagram socket to receive RTP packets
  from the server
  self.rtpSocket = socket.socket(socket.AF_INET, socket.
SOCK_DGRAM)

  # Set the timeout value of the socket to 0.5sec
  self.rtpSocket.settimeout(0.5)

  try:
    # Bind the socket to the address using the RTP port
given by the client user
    self.rtpSocket.bind(("", self.rtpPort))
  except:
```

```
        tkinter.messagebox.showwarning('Unable to Bind', '
    Unable to bind PORT=%d' % self.rtpPort)

```

- PLAY
  - Sent PLAY request. In this request, we must insert the Session header and use the session ID returned in the SETUP response but not put the Transport header.

```
elif requestCode == self.PLAY and self.state == self.READY:
        self.rtspSeq += 1
        request = 'PLAY ' + self.fileName + ' RTSP/1.0\n'+
                        'CSeq: ' + str(self.rtspSeq) + '\n'+\
                        'Session: ' + str(self.sessionId)
        self.requestSent = self.PLAY
```

  - Read the server's reponse

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

- PAUSE, TEARDOWN
  Send PAUSE, TEARDOWN request and parse server's response similar to PLAY.

```
    # Pause request
    elif requestCode == self.PAUSE and self.state == self.PLAYING
   :
      self.rtspSeq += 1
      request = 'PAUSE ' + self.fileName + ' RTSP/1.0\n'+\
                     'CSeq: ' + str(self.rtspSeq) + '\n'+\
                     'Session: ' + str(self.sessionId)
      self.requestSent = self.PAUSE
    # Teardown request
    elif requestCode == self.TEARDOWN and not self.state == self.
   INIT:
      self.rtspSeq += 1
      request = 'TEARDOWN ' + self.fileName + ' RTSP/1.0\n'+\
                     'CSeq: ' + str(self.rtspSeq) + '\n'+\
                     'Session: ' + str(self.sessionId)
      self.requestSent = self.TEARDOWN
```

```
elif self.requestSent == self.PAUSE:
    self.state = self.READY
    # The play thread exits. A new thread is created on resume.
    self.playEvent.set()
elif self.requestSent == self.TEARDOWN:
    self.state = self.INIT
  # Flag the teardownAcked to close the socket.
  self.teardownAcked = 1
```

3.3.2 Server

When the server receives the PLAY-request from the client, the server reads one video frame from the file and creates an RtpPacket-object which is the RTP-encapsulation of the video frame. It then sends the frame to the client over UDP every 50 milliseconds. For the encapsulation, the server calls the encode function of the RtpPacket class. We need to fill RTP packet header with the fields in Figure 3:

- RTP-version: V = 2
- P = X = CC = M (in this lab)
- Payload type: PT = 26 corresspond to MJPEG type.
- Sequence number = `frameNbr` where `frameNbr` is given by the server.
- Timestamp: use time module of Python
- SSRC is any integer
- CC = 0 (we have no other contributing sources. The CSRC field does not exist.

The total length of our RTP packet header is 12 bytes.

```
class RtpPacket:
  header = bytearray(HEADER_SIZE)

  def __init__(self):
    pass

  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)
    # Fill the header bytearray with RTP header fields
    # ...
    header[0] = header[0] | version << 6;
    header[0] = header[0] | padding << 5;
    header[0] = header[0] | extension << 4;
    header[0] = header[0] | cc;
    header[1] = header[1] | marker << 7;
    header[1] = header[1] | pt;

    header[2] = (seqnum >> 8) & 0xFF;
    header[3] = seqnum & 0xFF;

    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
    # ...
    self.payload = payload
```

3.4 Demonstration

First, we use command line to create RTSP port for server and RTP port for client.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\DELL\Desktop\submit> python Server.py 2048
```

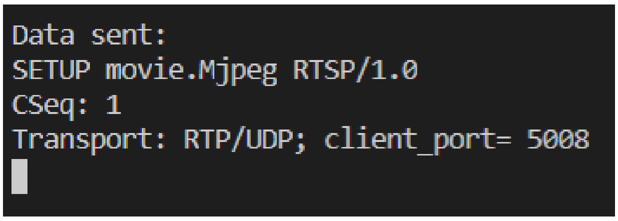
Hình 7: Create port for server

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\DELL\Desktop\submit> python ClientLauncher.py 127.0.0.1 2048 5008 movie.Mjpeg
```

Hình 8: Create port for client

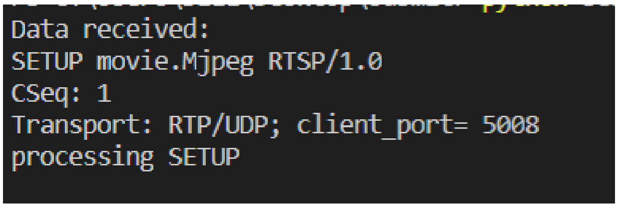
Hình 9: Initial user interface

Click on button 'SETUP'



```
Data sent:
SETUP movie.Mjpeg RTSP/1.0
CSeq: 1
Transport: RTP/UDP; client_port= 5008
```

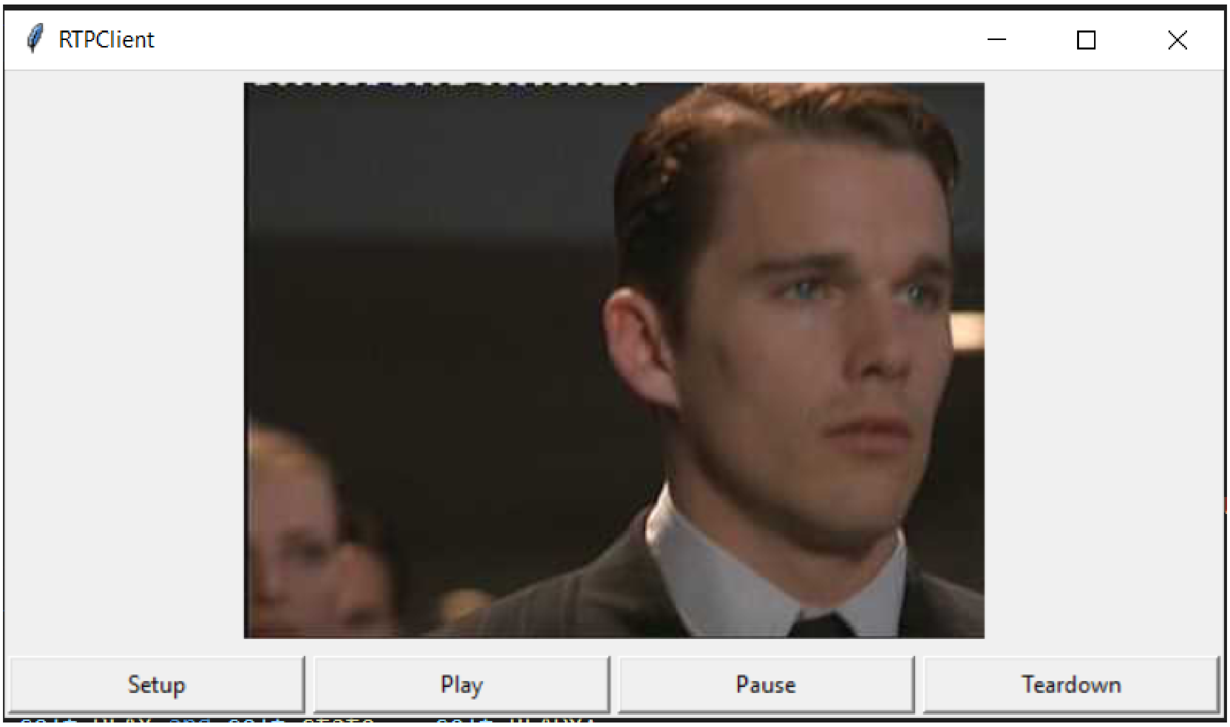
Hình 10: Client sent SETUP request



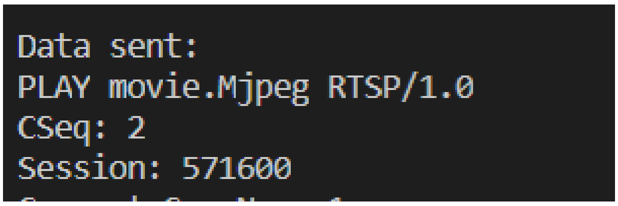
```
Data received:
SETUP movie.Mjpeg RTSP/1.0
CSeq: 1
Transport: RTP/UDP; client_port= 5008
processing SETUP
```

Hình 11: Server received SETUP request

Click on button 'PLAY'



Hình 12: User interface displays the video file



Hình 13: Client sent PLAY request

```
Data received:
PLAY movie.Mjpeg RTSP/1.0
CSeq: 2
Session: 571600
processing PLAY
```

Hình 14: Server received PLAY request

Click on button 'PAUSE'

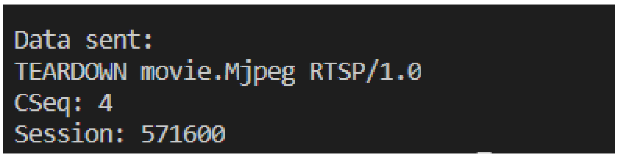
```
Data sent:
PAUSE movie.Mjpeg RTSP/1.0
CSeq: 3
Session: 571600
```

Hình 15: Client sent PAUSE request

```
Data received:
PAUSE movie.Mjpeg RTSP/1.0
CSeq: 3
Session: 571600
processing PAUSE
```

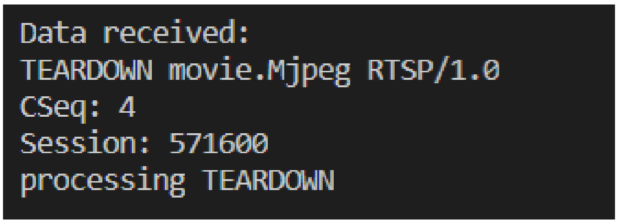
Hình 16: Server received PAUSE request

Click on button 'TEARDONW'



```
Data sent:
TEARDOWN movie.Mjpeg RTSP/1.0
CSeq: 4
Session: 571600
```

Hình 17: Client sent TEARDOWN request



```
Data received:
TEARDOWN movie.Mjpeg RTSP/1.0
CSeq: 4
Session: 571600
processing TEARDOWN
```

Hình 18: Server received TEARDOWN request

3.5 Evaluation

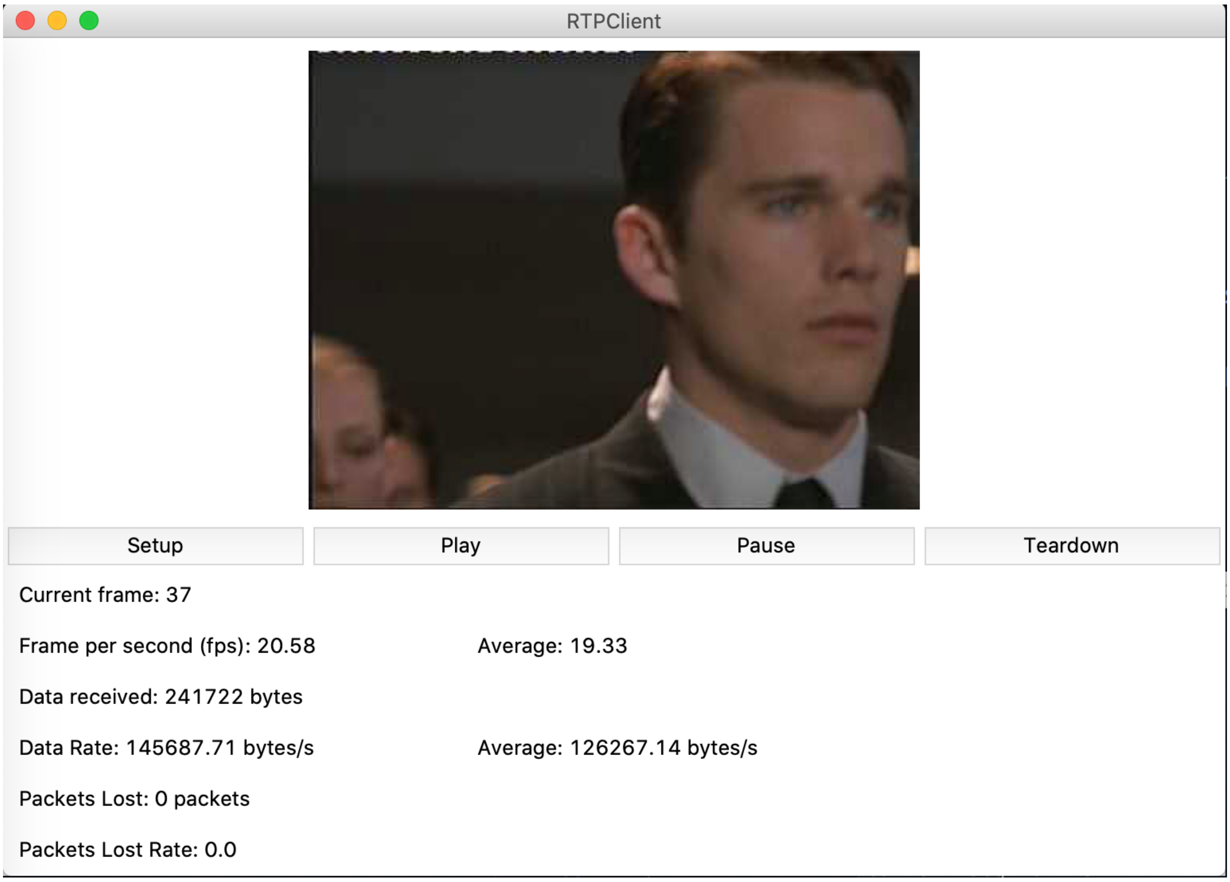
Task	Evaluation
Send RTSP request to server	100%
Open RTP socket binded to a specified port	100%
Encode the RTP packet with header fields and payload	100%
Display statistics of session	100%
Implement DESCRIBE request	100%

4 Extend

4.1 Extend 1

Statistics calculated about the sesion:

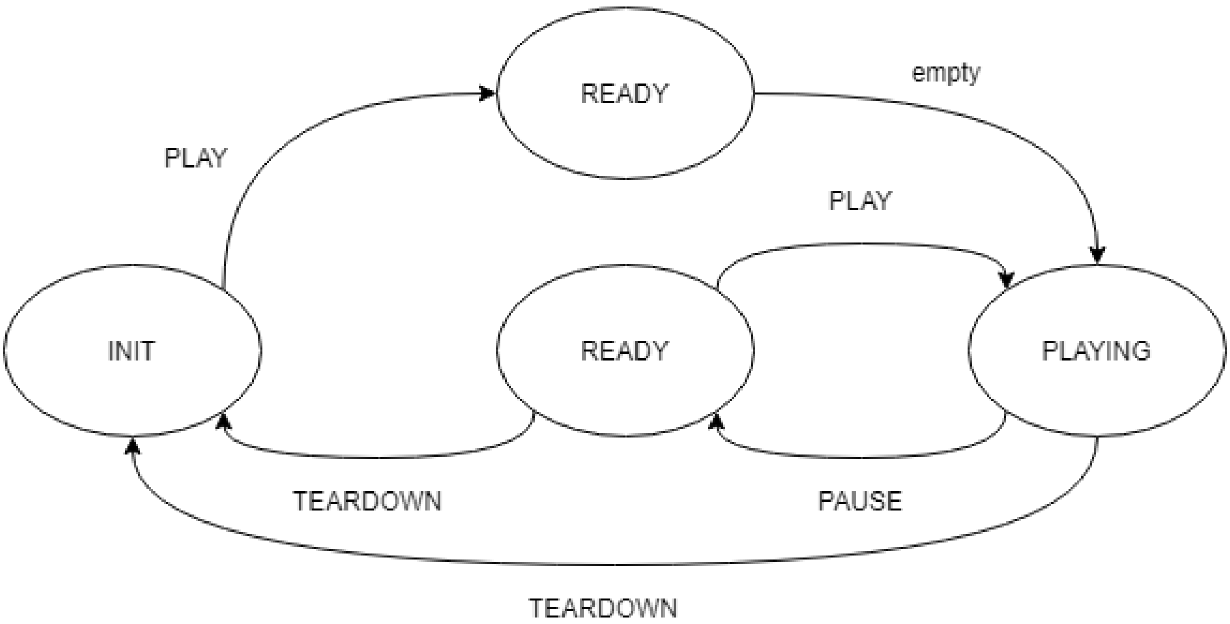
- Curent frame: number of displayed frame.
- Frame per second (fps): Number of frame displays in 1 second. (both instantaneous rate and avarage rate)
- Data recieved: Total data received.
- Data rate: Amount of data received in 1 second. (both instantaneous rate and avarage rate)
- Packets Lost: Number of packets lost up to the curent time.
- Packets Lost Rate: number of lost packets on number of received packets.



Hình 19: Some statistics about the session

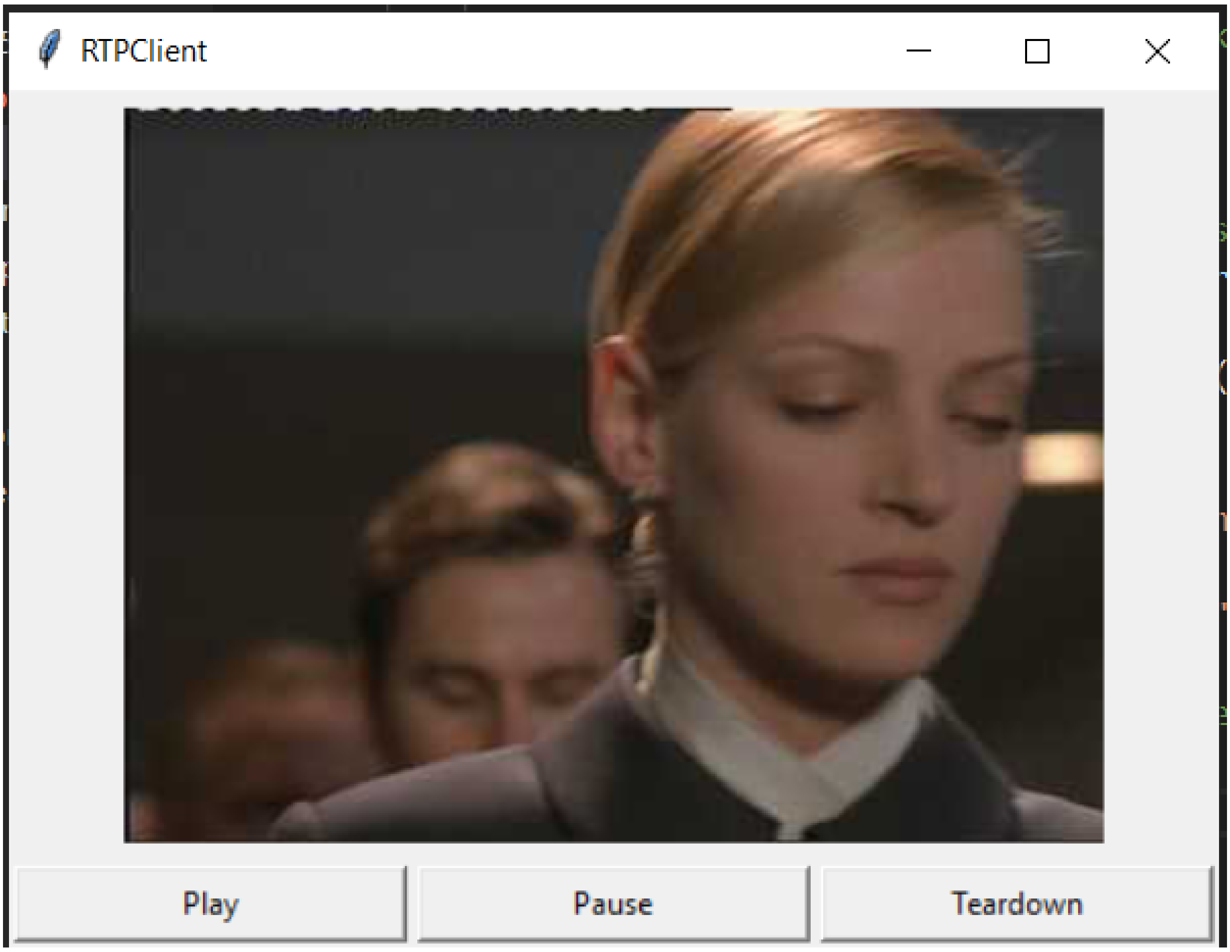
4.2 Extend 2

To solve the problem, we have implemented the PLAY button including the SETUP function. When pressing the PLAY button, the program will consider the client's current status as INIT or READY, in case of INIT the setup command will be executed before creating a new thread to send and receive RTP packets. Conversely, only new threads is created without excuting SETUP anymore.



Hình 20: Client States with 3 buttons: PLAY, PAUSE and TEARDOWN

```
def playMovie(self):
    """Play button handler."""
    wait = True
    if self.state == self.INIT:
        self.setupEvent = threading.Event()
        self.sendRtspRequest(self.SETUP)
        wait = self.setupEvent.wait(timeout=0.5)

    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)
```



Hình 21: User interface on RTPClient with 3 buttons

Servers are supposed to maintain a session state for a particular user. A server will not only receive requests from a single client port, but also from multiple clients at the same time. Therefore, for each client, when sending a SETUP request for the first time, the client is assigned with a unique session ID. All following requests sent from the client must include this session ID. By sending the TEARDOWN request, the client tells the server that it can release all state related to that user.

4.3 Extend 3

When the client sends a describe request to the server, in addition to sending the reply, the server also sends a session description, the client need to classify between the session description and the reply from server.
Session description:

Hình 22: User interface with DESCRIBE button

- v= (protocol version number, currently only 0)
- m= (media name and transport address)
- a=* (zero or more session attribute lines)
- a=* (zero or more media attribute lines — overriding the Session attribute lines)

We also have some other features, such as: content-base, content-type and content-length.

```
def sendDescription(self):
    description = '\nSession Description: \n\n'

```

```
    body = 'v=0' + CRLF
    body += 'm=video ' + str(self.clientInfo['server_port']) + ' RTP/AVP ' + MJPEG_TYPE + CRLF
    body += 'a=control:streamid=' + str(self.clientInfo['session']) + CRLF
    body += 'a=mimetype:string;\"video/MJPEG\"' + CRLF

    description += "Content-Base: " + self.clientInfo['videoFileName'] + CRLF
    description += "Content-Type: " + 'application/sdp' + CRLF
    description += 'Content-Length: ' + str(len(body)) + CRLF
    description += body

    connSocket = self.clientInfo['rtspSocket'][0]
    connSocket.send(description.encode())
```

For example, with a session 791628, server port 2048, rtp port 4096. The session description will be:

```
Data sent:
DESCRIBE movie.Mjpeg RTSP/1.0
CSeq: 4
Session: 791629

Session Description:

Content-Base: movie.Mjpeg
Content-Type: application/sdp
Content-Length: 90
v=0
m=video 2048 RTP/AVP 26
a=control:streamid=791629
a=mimetype:string;"video/MJPEG"
```

Hình 23: Session Description

5 Summary

In this assignment, we have successfully implemented 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). In addition, we also did entends about session statistics and session description, as well as compare the obtained results with the other standard media player.

Through the process of working on this assignemnt, we have practiced and used the knowledge we gained from the lectures. We also learned and understood more about how client and server communicate in an RTSP-Client-Server (how client sends request to server, and how server reply for this request).

6 References

1. Real Time Streaming Protocol
   Link: https://en.wikipedia.org/wiki/Real_Time_Streaming_Protocol

2. Session Description Protocol
   Link: https://en.wikipedia.org/wiki/Session_Description_Protocol?fbclid=IwAR3U6T3xF2g1ZjI2Aijf2nbW-g1mi6ROLWC4cRKjqiKci8yOy1whLu8a4MI

3. socket — Low-level networking interface
   Link: https://docs.python.org/3/library/socket.html

4. RTP, RTCP, and RTSP - Internet Protocols for Real Time Multimedia Communication, *Arjan Durreci, Raj jain*
   Link: https://www.cse.wustl.edu/~jain/books/ftp/rtp.pdf