ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

Project Socket Programming

Đề tài: Video Streaming with RTSP and RTP

Môn học: Mạng máy tính

Sinh viên thực hiện:

Nguyễn Quốc Nam - 24120098

Võ Hoàng Phúc - 24120123

Chế Nguyễn Thùy Trang - 24120469

Giáo viên hướng dẫn:

ThS. Chung Thuỳ Linh

Ngày 6 tháng 12 năm 2025

Danh sách thành viên

Đóng góp	MSSV	Họ và tên	Mã lớp	Nhiệm vụ
100%	24120123	Võ Hoàng Phúc	24CTT1	- Xây dựng code thỏa tiêu chí 1. - Tổng hợp tài liệu cho tiêu chí 1. - Đồng bộ code.
100%	24120098	Nguyễn Quốc Nam	24CTT1	- Xây dựng code thỏa tiêu chí 2. - Tổng hợp tài liệu cho tiêu chí 2. - Viết báo cáo.
100%	24120469	Chế Nguyễn Thùy Trang	24CTT1	- Xây dựng code thỏa tiêu chí 3. - Tổng hợp tài liệu cho tiêu chí 3.

Bảng 1: Danh sách các thành viên và nhiệm vụ

Mục lục

Danh sách bảng

Danh sách hình vẽ

1 Giới thiệu

1.1 Mục tiêu bài lab

Hiểu được và xây dựng được hệ thống streaming video dùng giao thức RTSP để điều khiển và RTP để truyền tải dữ liệu video từ server đến client.

1.2 Tiêu chí đánh giá

No.	Yêu cầu	Điểm
1	Triển khai giao thức RTSP của máy khách và đóng bóc gói tin RTP của máy chủ.	4đ
2	Truyền phát video HD	3đ
3	Bộ nhớ đệm phía máy khách	2.5đ
3	Báo cáo	0.5đ

Bảng 2: Tiêu chí chấm điểm bài lab

1.3 Tổng quan thư mục

Bài lab được tổ chức theo các thư mục sau:

- **1_RTSP_RTP**: Chứa source code cho tiêu chí 1.
- **2_HD**: Chứa source code cho tiêu chí 2.
- **3_CACHE**: Chứa source code cho tiêu chí 3.
- **4_REPORT**: Chứa báo cáo của bài lab (tiêu chí 4).
- **video**: Chứa các video với các độ phân giải khác nhau phù hợp với từng tiêu chí.

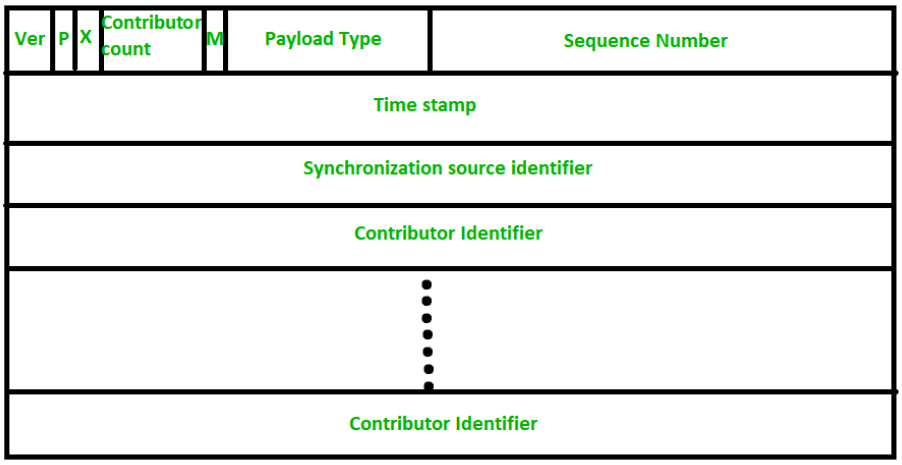
Phần source code cũng đã được up lên Github.

2 Cơ sở lý thuyết

2.1 RTP - Real time transfer protocol

RTP (Real-time Transfer Protocol) là giao thức truyền tải được thiết kế để vận chuyển dữ liệu thời gian thực như video, audio hoặc hình ảnh qua mạng IP. RTP thường hoạt động trên nền tảng UDP

để giảm độ trễ và cho phép truyền tải liên tục, mặc dù đặc tính này có thể dẫn đến mất gói (packet loss). [3]



Hình 1: Cấu trúc giao thức RTP [1]

Gói RTP bao gồm hai phần:

- Header: chứa các thông tin điều khiển quan trọng như:
  - Version: phiên bản của giao thức RTP.
  - Padding: chỉ định xem có thêm byte đệm vào cuối gói hay không.
  - Extension: chỉ định xem có phần mở rộng header hay không.
  - Contributing Source Count: số lượng nguồn đồng thời tham gia vào stream.
  - Marker Bit: đánh dấu các sự kiện quan trọng trong luồng dữ liệu.
  - Payload Type: xác định loại dữ liệu được truyền.
  - Sequence Number: xác định thứ tự gói tin, hỗ trợ phát hiện mất gói.
  - Timestamp: đánh dấu thời gian của gói tin, hỗ trợ đồng bộ hóa.
  - Synchronization Source Identifier: định danh nguồn stream.
  - Contributing Source Identifiers: định danh các nguồn đóng góp vào stream.
- Payload: chứa dữ liệu thực tế của video.

Lưu ý: Một số giá trị đã được quy định sẵn trong giới hạn bài lab

Ký hiệu	Field	Số bit	Giá trị
V	Version	2	2
P	Padding	1	0
X	Extension	1	0
CC	Contributing Source Count	4	0
M	Marker Bit	1	0
PT	Payload Type	7	26
	Sequence Number	16	frameNbr
	Timestamp	32	time()
SSRC	Synchronization Source Identifier	32	Số tùy ý
CSRC	Contributing Source Identifier	0	Không có
	Tổng số bits	**96 (12 bytes)**	

Bảng 3: Giá trị có sẵn của bài lab trong RTP header

2.2 RTSP - Real time streaming protocol

RTSP (Real-time Streaming Protocol) là giao thức điều khiển cho các phiên truyền tải media. Khác với RTP, RTSP không vận chuyển dữ liệu video mà cung cấp cơ chế "điều khiển từ xa" các hành động như PLAY, PAUSE hay TEARDOWN. RTSP hoạt động trên TCP để đảm bảo tính chính xác của thông điệp điều khiển. [4]

Một phiên RTSP tiêu chuẩn gồm các trạng thái và lệnh sau:

- **SETUP**: yêu cầu server tạo phiên làm việc RTSP và thiết lập port để truyền RTP.
- **PLAY**: yêu cầu server bắt đầu gửi các packet RTP tới client.
- **PAUSE**: tạm dừng việc gửi dữ liệu nhưng giữ phiên RTSP.
- **TEARDOWN**: yêu cầu server kết thúc phiên RTSP và giải phóng tài nguyên.

Client và server giao tiếp bằng mô hình request–response, trong đó mỗi yêu cầu đều kèm theo CSeq (Command Sequence) để đánh số thứ tự và Session ID để xác định phiên làm việc.

2.3 MJPEG - Motion JPEG

MJPEG (Motion JPEG) là định dạng video được tạo ra bằng cách nối liên tiếp nhiều ảnh JPEG độc lập theo thời gian để tạo thành chuỗi chuyển động. [5]

Mỗi frame được lưu dưới dạng ảnh JPEG đầy đủ với 1 trong các cấu trúc sau:

1. Trước mỗi frame có một 5-byte prefix mô tả độ dài frame đơn vị byte (file video có sẵn trong bài lab).

2. JPEG frame bắt đầu bằng `0xFF 0xD8` (SOI) và kết thúc bằng `0xFF 0xD9` (EOI) (các file video từ nguồn khác).

2.4 MTU - Maximum Transmission Unit

MTU (Maximum Transmission Unit) là kích thước tối đa của một đơn vị dữ liệu có thể được truyền đi trong một frame hoặc một gói tin trên một phương tiện mạng (network interface) mà không cần bị phân mảnh (fragmentation). [6]

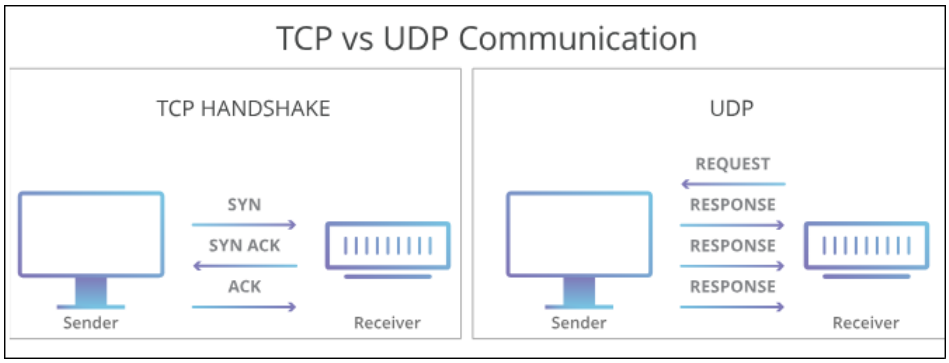
Các phương tiện truyền tải qua IP	**MTU (bytes)**	**Notes**
Internet IPv4 path MTU	Tối thiểu 68 bytes, tối đa 64 KiB	MTU thực tế phụ thuộc đường truyền; nếu router giảm MTU sẽ xảy ra IP fragmentation.
Internet IPv6 path MTU	Tối thiểu 1280 bytes, tối đa 64 KiB	Yêu cầu MTU tối thiểu 1280 bytes; hỗ trợ gói jumbo cho payload lớn.
X.25	576 bytes khi gửi hoặc 1600 bytes khi nhận	Chuẩn X.25 quy định hai ngưỡng MTU khác nhau cho chiều gửi và nhận.
Ethernet v2	1500 bytes	Hầu hết mạng LAN sử dụng Ethernet II nên MTU mặc định luôn là 1500 bytes.
...	...	...

Bảng 4: Một số MTU cho các phương tiện truyền tải phổ biến

Ngoài ra, WiFi (802.11) cũng dùng MTU 1500 cho IP payload và mọi router WiFi đều mặc định sử dụng MTU = 1500 để tương thích Internet. [7]

2.5 UDP - User datagram protocol và TCP - Transmission Control Protocol

TCP là giao thức truyền vận đáng tin cậy (reliable), hướng kết nối (connection-oriented), đảm bảo dữ liệu được gửi theo đúng thứ tự và không bị mất. UDP là giao thức truyền vận không kết nối (connectionless), không đảm bảo độ tin cậy (unreliable), không kiểm soát luồng dữ liệu và không đảm bảo thứ tự gói tin. [8]



Hình 2: So sánh TCP và UDP [2]

3 Quy trình thực hiện

3.1 Xây dựng RTP/RTSP server và client (4đ)

Các file trong phần này nằm trong thư mục **1_RTSP_RTP**

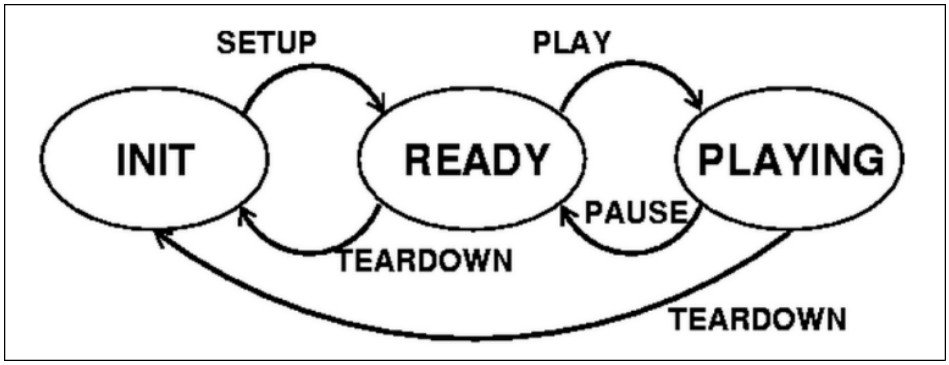
Đây là phiên bản đầu tiên cơ bản nhất của bài lab, yêu cầu xây dựng hệ thống streaming video sử dụng giao thức RTSP để điều khiển và RTP để truyền tải dữ liệu video từ server đến client.

3.1.1 Ý tưởng

Hoàn thiện giao thức RTSP trên client và đóng gói dữ liệu RTP trên server để truyền tải video từ server đến client trên các hàm đã được tạo sẵn.

3.1.2 Hoàn thiện Client.py - Giao thức RTSP của client

Mô hình trang thái của client:



Hình 3: Sơ đồ trạng thái và chuyển đổi của client RTSP

Hoàn thiện hàm gửi RTSP request - sendRtspRequest():

```
import threading

def sendRtspRequest(self, requestCode):
    if requestCode == self.SETUP and self.state == self.INIT:
        threading.Thread(target=self.recvRtspReply).start()
        self.rtspSeq += 1
        request = "SETUP " + str(self.fileName) + " RTSP/1.0\nCSeq: " + str(self.rtspSeq) + "\nTransport: RTP/UDP; client_port= " + str(self.rtpPort)
        self.requestSent = self.SETUP
    elif requestCode == self.PLAY and self.state == self.READY:
        self.rtspSeq += 1
        request = "PLAY " + str(self.fileName) + " RTSP/1.0\nCSeq: " + str(self.rtspSeq) + "\nSession: " + str(self.sessionId)
        self.requestSent = self.PLAY
    elif requestCode == self.PAUSE and self.state == self.PLAYING:
        self.rtspSeq += 1
        request = "PAUSE " + str(self.fileName) + " RTSP/1.0\nCSeq: " + str(self.rtspSeq) + "\nSession: " + str(self.sessionId)
        self.requestSent = self.PAUSE
    elif requestCode == self.TEARDOWN and not self.state == self.INIT:
        self.rtspSeq += 1
        request = "TEARDOWN " + str(self.fileName) + " RTSP/1.0\nCSeq: " + str(self.rtspSeq) + "\nSession: " + str(self.sessionId)
        self.requestSent = self.TEARDOWN
    else:
        return

    self.rtspSocket.send(request.encode())
    print('\nData sent:\n' + request)
```

- Hàm build chuỗi request. Mỗi lần gửi đều tăng `CSeq` để server đồng bộ và gắn `Session` để nhận biết sau `SETUP`. Cụ thể hơn:
  - `SETUP` request: Chương trình sẽ kiểm tra trạng thái hiện tại của client, nếu là `INIT` thì tạo request `SETUP` và bắt đầu chạy nền phương thức recvRtspReply (nhận các RTSP reply từ server). Một request `SETUP` sẽ có dạng:

```
SETUP <Video_File> RTSP/1.0
CSeq: <CSeq>
Transport: RTP/UDP; client_port= <RTP_Port>
```

Ví dụ:

```
SETUP video/movie288.Mjpeg RTSP/1.0
CSeq: 1
Transport: RTP/UDP; client_port= 8090
```

– `PLAY` request: Chương trình sẽ kiểm tra trạng thái hiện tại của client, nếu là `READY` thì tạo request `PLAY` có dạng:

```
PLAY <Video_File> RTSP/1.0
CSeq: <CSeq>
Session: <Session_ID>
```

Ví dụ:

```
PLAY video/movie288.Mjpeg RTSP/1.0
CSeq: 2
Session: 619227
```

– `PAUSE` request: Chương trình sẽ kiểm tra trạng thái hiện tại của client, nếu là `PLAYING` thì tạo request `PAUSE` có dạng:

```
PAUSE <Video_File> RTSP/1.0
CSeq: <CSeq>
Session: <Session_ID>
```

Ví dụ:

```
PAUSE video/movie288.Mjpeg RTSP/1.0
CSeq: 3
Session: 619227
```

– `TEARDOWN` request: Chương trình sẽ kiểm tra trạng thái hiện tại của client, nếu không là `INIT` (là `READY` hoặc `PLAYING`) thì tạo request `TEARDOWN` có dạng:

```
TEARDOWN <Video_File> RTSP/1.0
CSeq: <CSeq>
Session: <Session_ID>
```

Ví dụ:

```
TEARDOWN video/movie288.Mjpeg RTSP/1.0
CSeq: 4
Session: 619227
```

- Sau khi dựng xong, request được encode và đẩy qua socket TCP tới server RTSP.

Hoàn thiện hàm phân tích RTSP reply - parseRtspReply():

```
def parseRtspReply(self, data):
    lines = data.split('\n')
    seqNum = int(lines[1].split(' ')[1])

    if seqNum == self.rtspSeq:
        session = int(lines[2].split(' ')[1])
        if self.sessionId == 0:
            self.sessionId = session
        if self.sessionId == session:
            if int(lines[0].split(' ')[1]) == 200:
                if self.requestSent == self.SETUP:
                    self.state = self.READY
                    self.openRtpPort()
                elif self.requestSent == self.PLAY:
                    self.state = self.PLAYING
                elif self.requestSent == self.PAUSE:
                    self.state = self.READY
                    self.playEvent.set()
                elif self.requestSent == self.TEARDOWN:
                    self.state = self.INIT
                    self.teardownAcked = 1
```

- Phản hồi được tách theo từng dòng để lấy `CSeq` và `Session`. Client chỉ xử lý khi số thứ tự trùng với gói vừa gửi để tránh trạng thái cũ. Ví dụ một phản hồi có dạng:

```
RTSP/1.0 <HTTP_Status_Code> <Phrase>
CSeq: <CSeq>
Session: <Session_ID>
```

- Khi mã trạng thái HTTP là 200, client cập nhật trạng thái mới dựa trên request đã gửi:
  - Sau `SETUP`, trạng thái chuyển từ `INIT` sang `READY`. Sau đó mở port RTP bằng hàm openRtpPort().
  - Sau `PLAY`, trạng thái chuyển từ `READY` sang `PLAYING` và bắt đầu chạy nền hàm nhận gói RTP.
  - Sau `PAUSE`, trạng thái chuyển từ `PLAYING` sang `READY` và dừng hàm nhận gói RTP.
  - Sau `TEARDOWN`, trạng thái chuyển về `INIT` và đóng socket RTP.

Hoàn thiện hàm kết nối tới port của RTP - openRtpPort():

```
import socket, tkinter

def openRtpPort(self):
    self.rtpSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    self.rtpSocket.settimeout(0.5)

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

- Client khởi tạo socket UDP (`AF_INET, SOCK_DGRAM`) (AF_INET: IPv4, SOCK_DGRAM: UDP) cùng timeout ngắn để việc nhận khung không bị block.
- Hàm bind trực tiếp vào `rtpPort`; nếu cổng đã được sử dụng, chương trình sẽ cảnh báo không thể bind port đó.

3.1.3 Hoàn thiện RtpPacket.py - Giao thức RTP của server

Hoàn thiện hàm encapsulation - encode():

```
from time import time
HEADER_SIZE = 12

```

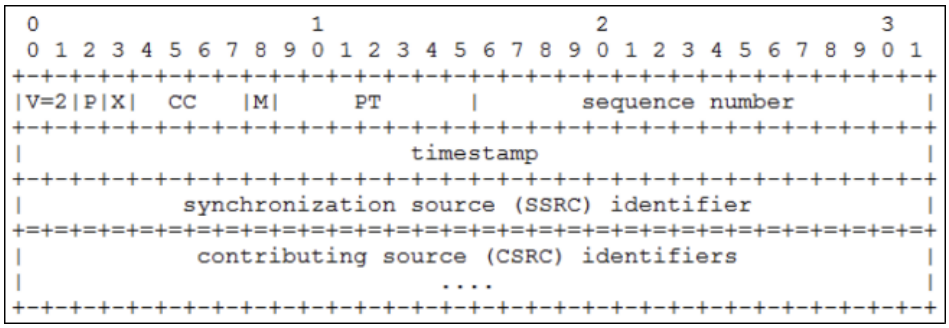
```
4  def encode(self, version, padding, extension, cc, seqnum, marker, pt, ssrc,
       payload):
5      timestamp = int(time())
6      header = bytearray(HEADER_SIZE)
7
8      header[0] = (version & 0x03) << 6 | (padding & 0x01) << 5 | (extension & 0
       x01) << 4 | (cc & 0x0F)
9      header[1] = (marker & 0x01) << 7 | (pt & 0x7F)
10     header[2:4] = (seqnum >> 8) & 0xFF, seqnum & 0xFF
11     header[4:8] = (timestamp >> 24) & 0xFF, (timestamp >> 16) & 0xFF, (timestamp
       >> 8) & 0xFF, timestamp & 0xFF
12     header[8:12] = (ssrc >> 24) & 0xFF, (ssrc >> 16) & 0xFF, (ssrc >> 8) & 0xFF,
       ssrc & 0xFF
13
14     self.header = header
15     self.payload = payload
```

header: mảng byte có kích thước 12 bytes

- Byte thứ 1 (8 bits): chứa trường Version, Padding, Extension, Contributing Source Count và có dạng V-V-P-X-CC-CC-CC-CC
  - version & 0x03: lấy 2 bit cuối của version (03[16] = 0000 0011[2])
  - « 6: dịch trái 6 bit để đưa về vị trí 2 bit đầu tiên trong byte (eg. V = 2 => 1100 0000[2])
  - Tương tự với các trường padding, extension, lúc này còn 4 bit cuối cùng là CC
  - cc & 0x0F: chỉ lấy 4 bit cuối của cc (0F[16] = 0000 1111[2])
- Byte thứ 2 (8 bits): chứa trường Marker Bit, Payload Type và có dạng M-PT-PT-PT-PT-PT-PT-PT (các logic tương tự byte 1)
- Byte thứ 3-4 (16 bits): chỉ chứa trường Sequence Number
  - (seqnum » 8) & 0xFF: lấy 8 bit đầu của seqnum (0xFF[16] = 1111 1111[2])
  - seqnum & 0xFF: lấy 8 bit cuối của seqnum
- Byte thứ 5-8 (32 bits): chỉ chứa trường Timestamp, logic tương tự byte 3-4

- Byte thứ 9-12 (32 bits): chỉ chứa trường Synchronization Source Identifier (SSRC), logic tương tự byte 3-4

Phần header trong python sẽ có dạng như sau, với mỗi dòng tương ứng với 4 bytes (32 bits):



Hình 4: Cấu trúc mảng header

3.1.4 Thử nghiệm

Thực hiện chạy server RTSP/RTP và client RTSP trên cùng máy tính với video mẫu qua câu lệnh:

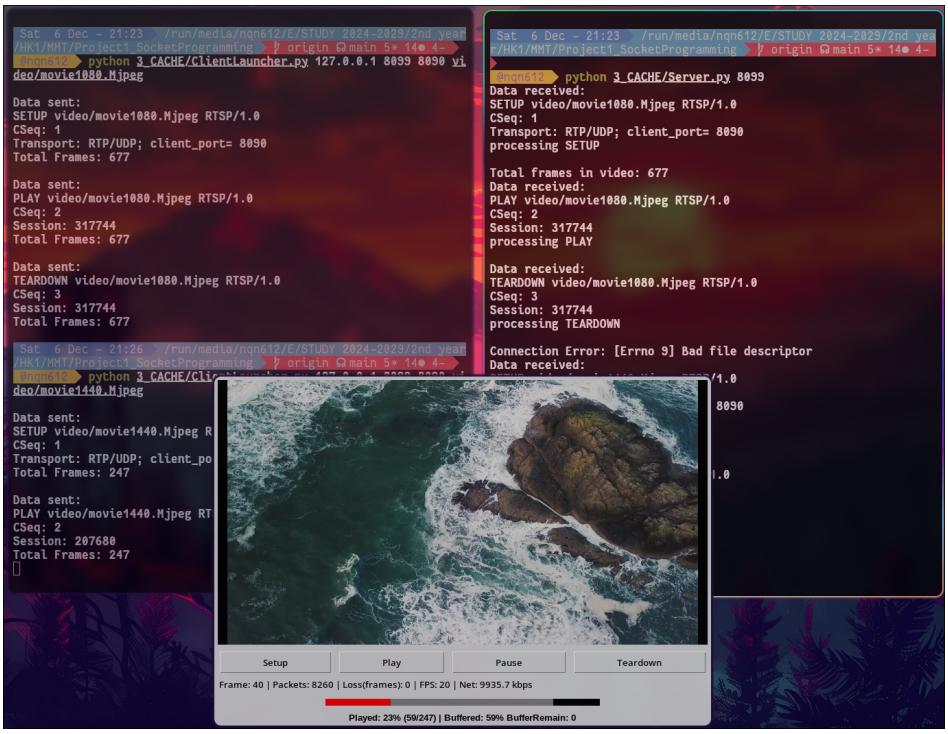
- Ternimal 1: chạy server RTSP/RTP (python Server.py <**PORT**>). Ví Dụ:

```
python 1_RTSP_RTP/Server.py 8089
```

- Ternimal 2: chạy client RTSP (python ClientLauncher.py <**SERVER_IP**> <**RTSP_PORT**> <**RTP_PORT**> <**VIDEO_FILE**>). Ví Dụ:

```
python 1_RTSP_RTP/ClientLauncher.py 127.0.0.1 8089 8090 video/movie288.
    Mjpeg
```

Output thu được khi chọn các nút (Setup -> Play -> Pause) trên giao diện client:



Hình 5: Output RTSP/GUI client tiêu chí 1

Có thể thấy client đã gửi đúng các request RTSP và chạy được video qua giao thức RTP.

Nhược điểm: Với video có độ phân giải cao (HD trở lên) khi 1 frame vượt quá kích thước gói UDP sẽ bị tràng byte qua các gói khác, dẫn đến việc giải mã sai các frame tiếp theo và báo lỗi.

3.2 Bổ sung HD video streaming (3đ)

Các file trong phần này nằm trong thư mục **2_HD_Video_Streaming**.

Đây là phiên bản nâng cấp của phần 1 nhằm hỗ trợ streaming video có độ phân giải cao (HD - 720p, Full HD - 1080p).

3.2.1 Ý tưởng

Khắc phục nhược điểm của phần 1 bằng cách chia nhỏ frame video thành các mảnh (fragment) có kích thước phù hợp với gói UDP dựa trên MTU (1500 bytes).

Mỗi fragment sẽ được đóng gói trong một gói RTP riêng biệt (max 20Kb) với đầy đủ header và payload. Client sẽ nhận các gói RTP, tái tạo lại các fragment và ghép chúng lại thành frame hoàn

chỉnh bằng cách nhận diện các SOI, EOI trong payload (Marker Bit không được sử dụng do đã bị giới hạn giá trị bằng 0).

3.2.2 Hoàn thiện cơ chế chia nhỏ frame thành các fragment

Trước hết để xử lý các video HD không có prefix cho biết độ dài frame, ta cần thêm một số phương thức trong file **VideoStream.py**:

```
def detectLengthPrefix(self):
    pos = self.file.tell()
    peek = self.file.read(5)
    self.file.seek(pos)
    return peek.isdigit()
```

Phương thức này để kiểm tra xem file video có sử dụng length prefix (5 bytes đầu chứa số độ dài frame) hay không. Nếu có, ta sẽ sử dụng phương thức cũ để đọc frame. Nếu không, ta sẽ sử dụng phương thức bên dưới.

```
def loadRawFrames(self):
    content = self.file.read()
    frames = []
    start = content.find(b'\xff\xd8')
    while start != -1:
        end = content.find(b'\xff\xd9', start + 2)
        if end == -1:
            break
        frames.append(content[start:end + 2])
        start = content.find(b'\xff\xd8', end + 2)
    return frames
```

Phương thức này sẽ lưu các frame video vào một danh sách (danh sách này sẽ được lưu bên phía server) bằng cách tìm kiếm các byte SOI (Start of Image - `0xFFD8`) và EOI (End of Image - `0xFFD9`) trong file video. Mỗi khi tìm thấy một cặp SOI và EOI, ta sẽ trích xuất dữ liệu giữa chúng và lưu vào danh sách frames.

```
def nextFrame(self):
    if self.lengthPrefixed:
        data = self.file.read(5)
        if data:
```

```
            framelength = int(data)
            data = self.file.read(framelength)
            self.frameNum += 1
        return data
    else:
        if self.frameNum < len(self.rawFrames):
            data = self.rawFrames[self.frameNum]
            self.frameNum += 1
            return data
        return None
```

Phương thức này sẽ trả về frame tiếp theo tùy theo loại Mjpeg của video và cộng thêm 1 cho chỉ số thứ tự frame hiện tại:

- Nếu video có length prefix, ta sẽ đọc 5 bytes đầu để lấy độ dài frame - `framelength`, sau đó đọc `framelength` tiếp theo là dữ liệu frame.
- Nếu video không có length prefix, ta sẽ lấy frame từ danh sách frames đã lưu trước đó.

Tiếp theo, ta sẽ chỉnh sửa hàm gửi RTP server phía server trong **ServerWorker.py** để chia nhỏ frame thành các fragment và gửi từng fragment trong một gói RTP riêng biệt:

```
MAX_PAYLOAD = 1500

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

        if self.clientInfo['event'].isSet():
            break
        data = self.clientInfo['videoStream'].nextFrame()
        if data:
            try:
                frameNumber = self.clientInfo['videoStream'].frameNbr()
                address = self.clientInfo['rtspSocket'][1][0]
                port = int(self.clientInfo['rtpPort'])
                for start in range(0, len(data), MAX_PAYLOAD):
                    if self.clientInfo['event'].isSet():
                        break
```

```
                    fragment = data[start:start + MAX_PAYLOAD]
                    self.seqNum += 1
                    packet = self.makeRtp(fragment, self.seqNum, frameNumber)
                    self.clientInfo['rtpSocket'].sendto(packet, (address, port))
            except:
                print("Connection Error")
```

Việc gửi RTP sẽ được thực hiện liên tục trong vòng lặp cho đến khi nhấn `PAUSE`, cụ thể:

1. Lấy 1 frame tiếp theo bằng hàm `nextFrame()`.
2. Chia frame này thành các fragment có kích thước `MAX_PAYLOAD` (1500 bytes) để đảm bảo không vượt quá giới hạn của gói UDP.
3. Mỗi fragment sẽ được đóng gói trong một gói RTP riêng biệt với đầy đủ header và payload bằng hàm `RtpPacket.encode()`.
4. Gửi từng gói RTP qua socket UDP tới client.

Cuối cùng, ta sẽ chỉnh sửa hàm nhận RTP client và thêm một số hàm cần thiết trong **Client.py** để tái tạo lại các fragment và ghép chúng thành frame hoàn chỉnh:

```
SOI = b'\xff\xd8'
EOI = b'\xff\xd9'
def tryAssembleFrame(self):
    while True:
        start = self.frameBuffer.find(SOI)
        if start == -1:
            self.frameBuffer.clear()
            break
        end = self.frameBuffer.find(EOI, start + 2)
        if end == -1:
            if start > 0:
                del self.frameBuffer[:start]
            break
        frame = self.frameBuffer[start:end + 2]
        del self.frameBuffer[:end + 2]
        self.updateMovie(self.writeFrame(frame))
```

Sau khi nhận được một gói RTP, ta sẽ trích xuất payload và lưu vào `frameBuffer` và gọi hàm `tryAssembleFrame()` để kiểm tra xem trong `frameBuffer` đã có đủ một frame hoàn chỉnh chưa. Nếu có, ta sẽ ghép các fragment lại thành frame hoàn chỉnh và render lên giao diện.

3.2.3 Hoàn thiện cơ chế bỏ qua các frame bị trễ để đảm bảo đồ mượt khi phát video

Trong quá trình nhận gói RTP, do tính chất không kết nối của UDP và việc chia nhỏ frame thành các fragment, có thể xảy ra tình trạng mất gói hoặc trễ gói. Điều này dẫn đến việc client không thể tái tạo đúng frame hoàn chỉnh từ các fragment nhận được. Để khắc phục vấn đề này và đảm bảo tính mượt mà khi phát video, ta sẽ bổ sung cơ chế bỏ qua các frame bị trễ dựa trên số thứ tự frame trong phương thức `listenRtp.py` trong **Client.py**:

```
data = self.rtpSocket.recv(20480)
if data:
    self.packetCount += 1
    self.bytesReceived += len(data)
    rtpPacket = RtpPacket()
    rtpPacket.decode(data)
    frameId = rtpPacket.timestamp()
    print("Current Frame Num: " + str(frameId))
    if frameId < self.frameNbr:
        continue
    if frameId > self.frameNbr:
        if self.frameNbr >= 0 and frameId - self.frameNbr > 1:
            self.frameLoss += frameId - self.frameNbr - 1
        self.frameNbr = frameId
        self.frameBuffer.clear()
    self.frameBuffer.extend(rtpPacket.getPayload())
    self.tryAssembleFrame()
```

ameNbr Hàm sẽ kiểm tra các điều kiện sau:

- Nếu số thứ tự frame nhận được (`frameId`) < số thứ tự frame đang chờ xử lý (`frameNbr`) → frame này đã bị trễ và không còn cần thiết ⇒ bỏ qua frame này.
- Nếu số thứ tự frame nhận được (`frameId`) > số thứ tự frame đang chờ xử lý (`pendingFrameId`) → có frame bị mất ⇒ tăng biến đếm `frameLoss` để theo dõi số frame bị mất.

- Chỉ khi số thứ tự frame nhận được (`frameId`) = số thứ tự frame đang chờ xử lý (`pendingFrameId`) ⇒ ghép các fragment thành frame hoàn chỉnh và hiển thị lên giao diện.

Ngoài ra hàm cũng sẽ in ra terminal frame hiện tại và lặp lại cho đến khi qua frame tiếp theo (số dòng lặp lại là số fragment của frame đó).

3.2.4 Hoàn thiện tính năng hiện số frame mất và lưu lượng mạng trên giao diện

Các thông số này được cập nhật trong hàm `updateStats.py` trong **Client.py**:

```
def updateStats(self):
    now = time.time()
    elapsed = now - self.startTime if self.startTime else 0
    fps = len(self.frameTimes)
    throughput = (self.bytesReceived * 8 / 1000) / elapsed if elapsed > 0 else 0
    stats = (
        f"Frame: {self.frameNbr} "
        f"| Packets: {self.packetCount} "
        f"| Loss(frames): {self.frameLoss} "
        f"| FPS: {fps:.2f} "
        f"| Net: {throughput:.2f} kbps"
    )
    self.statLabel.configure(text=stats)
    self.frameTimes.clear()
```

- Số frame nhận được (`frameNbr`): được cập nhật mỗi khi hiển thị một frame mới lên giao diện trong hàm `tryAssembleFrame()`.
- Số packet nhận được (`packetNbr`): được tăng trong hàm `listenRtp()` mỗi khi nhận được một gói RTP.
- Số frame mất (`frameLoss`): được tăng trong hàm `listenRtp()` mỗi khi phát hiện có frame bị mất.
- Lưu lượng mạng (Network Throughput): được tính bằng tổng số byte nhận được chia cho thời gian đã trôi qua kể từ khi bắt đầu nhận gói RTP.
- FPS được tính bằng số frame đã trôi qua trong 1 giây trước đó.

Ngoài ra, container chứa video sẽ scale theo kích thước cửa sổ để hiển thị video HD không bị crop và dễ nhìn hơn.

3.2.5 Thử nghiệm

Thực hiện chạy server RTSP/RTP và client RTSP trên cùng máy tính với video mẫu qua câu lệnh tương tự phần 1. Ví Dụ:

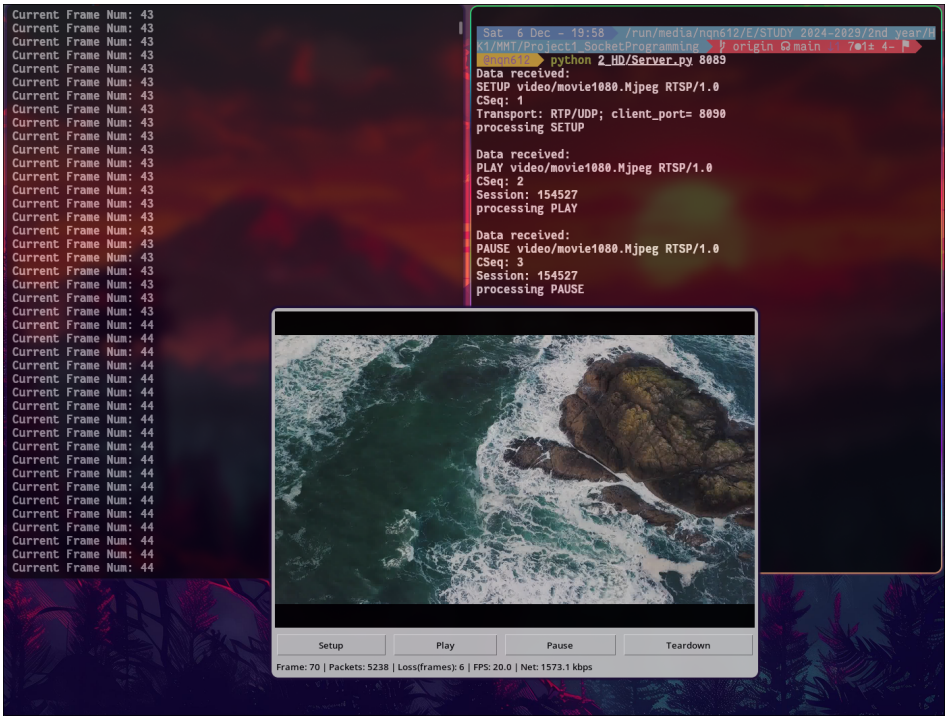
- Ternimal 1: chạy server RTSP/RTP

```
python 2_HD/Server.py 8089
```

- Ternimal 2: chạy client RTSP

```
python 2_HD/ClientLauncher.py 127.0.0.1 8089 8090 video/movie1080.Mjpeg
```

Output thu được trên giao diện client:



Hình 6: Output RTSP/GUI client tiêu chí 2

- Có thể thấy ở terminal của server, số frame được gửi được in thành nhiều dòng, nghĩa là frame đó đã được chia thành các fragment và gửi đi.

- Trên giao diện client cũng đã gồm đầy đủ các thông tin về số frame, packet thu được, frame mất, fps, lưu lượng mạng.
- Các video Mjpeg dù thuộc định dạng nào cũng đã chạy ổn định ở 20fps.

Nhược điểm: Dù đã gửi và chạy được nhưng với các video có độ phải cao hơn (1440p, 2160p,...) các frame tải chậm hơn tốc độ của video dẫn tới tình trạng mất frame và drop FPS hoặc tệ hơn là video bị đứng.

3.3 Bổ sung client cache (2.5đ)

Các file trong phần này nằm trong thư mục **3_Client_Cache**.
Đây là phiên bản nâng cấp của phần 2 và cũng là phiên bảng cuối cùng nhằm cải thiện trải nghiệm xem video HD bằng cách thêm bộ đệm (buffer) trên client.

3.3.1 Ý tưởng

Khắc phục nhược điểm của phần 2 bằng cách thêm bộ đệm (buffer) trên client để lưu trữ trước một số frame video nhất định trước khi phát. Khi nhấn Play, client sẽ bắt đầu nhận các gói RTP và lưu trữ các frame vào bộ đệm cho đến khi đạt ngưỡng đã định sẵn (ví dụ: 30 frame). Sau đó, client mới bắt đầu phát video từ bộ đệm trong khi tiếp tục nhận và lưu trữ các frame mới vào bộ đệm. Điều này giúp đảm bảo rằng có đủ frame để phát liên tục ngay cả khi tốc độ tải về chậm hơn tốc độ phát video.

3.3.2 Hoàn thiện cơ chế bộ đệm phía client

Để giải quyết tình trạng mất frame, giật hình, drop FPS hoặc đứng hình khi phát video độ phân giải cao (1080p, 1440p, 2160p), phía client được bổ sung một cơ chế bộ đệm (buffer) trong **Client.py** như sau:

```
def enqueueFrame(self, frameId, frameData):
    if not frameData or len(frameData) < 100:
        return

    with self.queueLock:
        self.frameQueue.append((frameId, frameData))
```

```
        self.frameBuffer.append(frameId)

        if len(self.frameQueue) > self.MIN_BUFFER_SIZE:
            self.frameQueue.popleft()

        if len(self.frameBuffer) > self.MAX_BUFFER_SIZE:
            self.frameBuffer.pop(0)

    self.bufferEvent.set()
```

Phương thức này sẽ thực hiện các bước sau:

1. Kiểm tra frame hợp lệ.
2. Thêm frame vào `frameQueue` (queue chính phục vụ playback).
3. Đồng thời thêm ID vào `frameBuffer` để theo dõi dung lượng buffer.
4. Nếu queue vượt quá `MIN_BUFFER_SIZE`, client tự động loại frame cũ nhất → tránh lag dồn.
5. Nếu buffer vượt `MAX_BUFFER_SIZE`, loại frame buffer cũ → giữ mức tối ưu.

Một số hằng số được định nghĩa để điều khiển cơ chế bộ đệm:

- `MIN_BUFFER_SIZE` = 20: yêu cầu phải có tối thiểu 20 frame trước khi bắt đầu phát → tránh giật khi bắt đầu video.
- `MAX_BUFFER_SIZE` = 100: client không cho phép buffer quá lớn để tránh tràn bộ nhớ.
- `deque()`: cấu trúc hàng đợi tốc độ cao dùng để chứa frame hoàn chỉnh.

Giữ tốc độ phát video ổn định ngay cả khi mạng có độ trễ hoặc server gửi chậm (tránh tràn bộ đệm bằng cách đặt giới hạn tối đa) bằng phương thức `playbackLoop` trong **Client.py**:

```
def playbackLoop(self):
    nextFrameDeadline = time.time()
    while not self.displayEvent.is_set():
        frame = None
        with self.queueLock:
            if self.frameQueue:
```

```
                frame = self.frameQueue.popleft()
        if frame:
            frameId, frameData = frame
            self.frameNbr = frameId
            self.framesDisplayed += 1
            self.playedFrames += 1
            self.updateMovie(frameData)
            self.updateStatsLabel()
            nextFrameDeadline = max(time.time(), nextFrameDeadline)
            sleepTime = max(0, nextFrameDeadline - time.time())
        else:
            self.bufferEvent.wait(TARGET_FRAME_INTERVAL)
            self.bufferEvent.clear()
            sleepTime = TARGET_FRAME_INTERVAL
        time.sleep(sleepTime)
        nextFrameDeadline = time.time() + TARGET_FRAME_INTERVAL
```

Phương thức này sẽ thực hiện các bước sau:

1. Nếu queue có frame → phát ngay, đảm bảo không bị trễ bởi thời gian nhận từ server.
2. Nếu queue rỗng → chờ frame mới bằng bufferEvent.wait().
3. `TARGET_FRAME_INTERVAL` = 1/20 giúp luôn phát đều 20 FPS → video mượt ổn định.
4. Không có block nào phụ thuộc vào tốc độ nhận gói mạng → playback không bị giật.

3.3.3 Thử nghiệm

Thực hiện chạy server RTSP/RTP và client RTSP trên cùng máy tính với video mẫu qua câu lệnh tương tự phần 1. Ví Dụ:

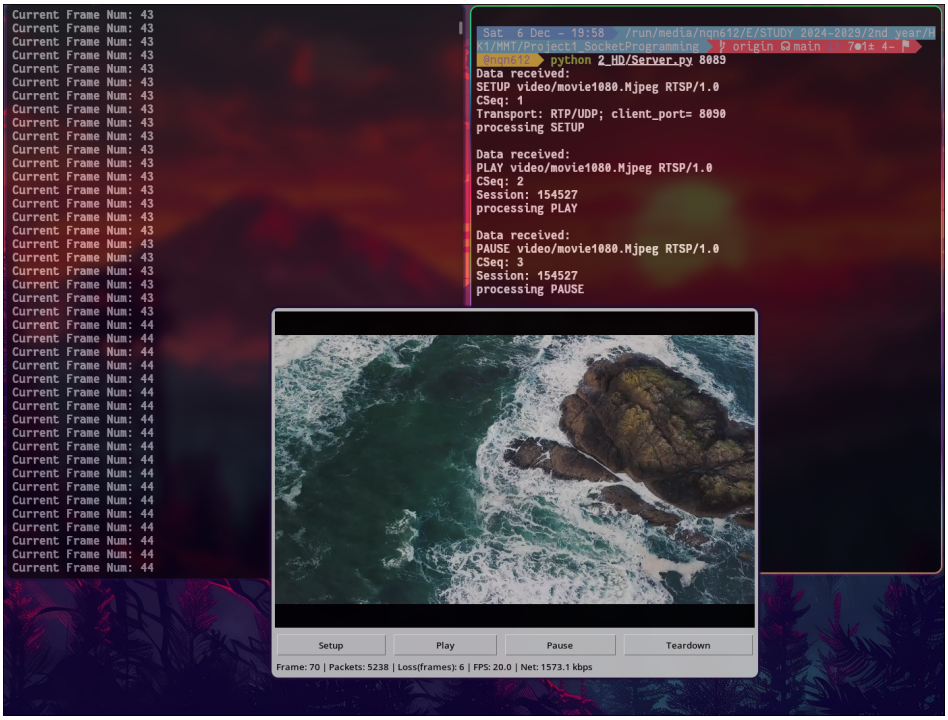
- Ternimal 1: chạy server RTSP/RTP

```
python 3_CACHE/Server.py 8089
```

- Ternimal 2: chạy client RTSP

```
python 3_CACHE/ClientLauncher.py 127.0.0.1 8089 8090 video/movie1440.Mjpeg
```

Output thu được trên giao diện client:



Hình 7: Output RTSP/GUI client tiêu chí 3

Có thể thấy video 1440p đã chạy mượt mà ở 20 FPS mà không bị giật hình hay drop frame, ngay cả khi mạng có độ trễ nhẹ. Thanh play bar và buffer bar cũng phản hồi đúng với thời lượng video.

Lợi ích của cơ chế Client-Side caching:

- Video ổn định hơn trong khi mạng chưa kịp ổn định
- Giảm giật hình do trễ UDP
- Hỗ trợ phát lại và đọc nhanh frame từ file
- Giảm tải cho server khi client không yêu cầu gửi lại các frame đã có

Client-side caching làm cho hệ thống RTP/RTSP trở nên mượt mà và gần với mô hình của các nền tảng streaming chuyên nghiệp. Cơ chế queue + file cache giúp đảm bảo tính ổn định và liên tục khi mạng có độ trễ hoặc mất gói

4 Kết quả đạt được

Sau quá trình xây dựng, hoàn thiện và thử nghiệm hệ thống theo ba tiêu chí của bài tập lớn, nhóm đã thu được các kết quả như sau.

4.1 Chức năng RTSP/RTP

Client có khả năng gửi các yêu cầu RTSP và nhận phản hồi đúng định dạng từ server. Phía server có thuật toán đóng gói RTP được hoàn thiện với đầy đủ các trường trong header. Hệ thống đã có thể truyền tải được các video MJPEG có độ phân giải thấp qua UDP với tốc độ ổn định và độ trễ thấp.

4.2 Chức năng truyền tải video HD

Chia nhỏ mỗi frame MJPEG thành các fragment có kích thước phù hợp với MTU. Đảm bảo tốc độ truyền ổn định và tốc độ phát mượt. Hiện thị được số lượng frame mất và lưu lượng mạng. Hệ thống đã có thể phát được video HD với tốc độ ổn định.

4.3 Chức năng bộ đệm phía client

Client luôn duy trì một số lượng frame được lưu trước (pre-buffer), đồng thời bổ sung các frame mới trong quá trình phát nhằm cải thiện khả năng xem lại và hạn chế giật hình khi video có độ phân giải lớn. Hệ thống đã có thể phát video có độ phân giải cao hơn như 1440p, 2160p mà không bị drop frame khi mạng không ổn định.

5 Tổng kết

Hệ thống cuối cùng đáp ứng toàn bộ yêu cầu của bài lab, từ việc thực thi giao thức RTSP và đóng gói RTP, đến việc truyền tải video HD và hỗ trợ client cache. Các chức năng đều hoạt động như mong đợi, thể hiện sự phù hợp của thiết kế hệ thống cũng như cách tiếp cận từng bước của nhóm.

6 Tài liệu tham khảo

Tài liệu

[1] Geeksforgeeks. Real-time Transport Protocol(RTP), 12-07-2025.

[2] Cloudflare. What is UDP?, ?-?-?

[3] Wikipedia. Real-time Transport Protocol, 11-12-2022.

[4] Wikipedia. Real-Time Streaming Protocol, 14-02-2022.

[5] Wikipedia. Motion JPEG, 21-11-2018.

[6] Wikipedia. Maximum Transmission Unit, 02-09-2021.

[7] IEFT - RFC 948. A Standard for the Transmission of IP Datagrams over IEEE 802 Networks, 02-1988.

[8] Cloudflare. What is TCP/IP?, ?-?-?