Project Socket Programming
Video Streaming with RTSP and RTP

I. Lời nói đầu

Trong đồ án này, chúng em tiến hành xây dựng một hệ thống truyền phát video theo thời gian thực (streaming video) bao gồm máy chủ (server) và máy khách (client). Hai thành phần này giao tiếp với nhau thông qua giao thức RTSP (Real-Time Streaming Protocol), đồng thời dữ liệu video được truyền tải bằng giao thức RTP (Real-time Transfer Protocol).

Thông qua đồ án này, chúng em hiểu rõ nguyên lý hoạt động của các giao thức truyền thông đa phương tiện trong môi trường mạng. Cụ thể, nhiệm vụ của chúng em là cài đặt giao thức RTSP ở phía client và thực hiện cơ chế đóng gói các gói tin RTP ở phía server.

Trong khuôn khổ đồ án này, giảng viên đã cung cấp sẵn mã nguồn xử lý RTSP ở phía server, giải đóng gói RTP ở phía client, cũng như chức năng hiển thị video nhận được. Dựa trên nền tảng đó, chúng em tập trung triển khai các phần còn thiếu nhằm hoàn thiện hệ thống truyền phát video, từ đó nắm vững hơn cách thức hoạt động của RTSP và RTP trong các ứng dụng truyền thông thời gian thực.

Mục lục

II. Giới thiệu hệ thống

1. Mục tiêu của đồ án

- Đồ án **"Video Streaming với RTSP và RTP"** xây dựng một hệ thống **client-server** cho phép phát video **MJPEG** qua mạng theo thời gian thực. Hệ thống sử dụng 2 kênh chính:

- **RTSP (Real-Time Streaming Protocol)** làm *kênh điều khiển*: client gửi lệnh điều khiển như **SETUP, PLAY, PAUSE, TEARDOWN** đến server.
- **RTP (Real-Time Transport Protocol)** làm *kênh truyền tải media*: server đóng gói từng khung hình **JPEG** thành gói **RTP** và gửi đến client qua **UDP**.

- Mục tiêu chính là:

1. Cài đặt **giao thức RTSP ở phía client** (gửi request, nhận & phân tích response, quản lý trạng thái phiên (INIT, READY, PLAYING), xử lý các nút bấm **Setup / Play / Pause / Teardown** trên GUI).
2. Cài đặt **đóng gói dữ liệu video thành gói RTP ở phía server** (packetization – đọc file MJPEG, tách từng frame, gắn header RTP (sequence number, timestamp, payload type, SSRC, ...) và gửi qua UDP tới đúng cổng RTP của client).
3. Mở rộng thêm tính năng **HD Video Streaming (720p/1080p)**: phân mảnh frame lớn, phát video mượt với độ trễ thấp, và thống kê **Frame loss rate** và **Video data rate (bytes/sec).**
4. Client-Side Caching (frame buffer ở phía client): Sau khi **SETUP** thành công, client khởi tạo một **frame buffer**. Mỗi lần người dùng nhấn **PLAY**, client sẽ **pre-buffer** trước N frame (trong code N = 10) vào **buffer** rồi mới bắt đầu hiển thị; các frame tiếp theo được lấy ra từ **buffer** theo cơ chế **FIFO**. Cách này giúp giảm **jitter** và làm cho việc phát video HD mượt hơn.

2. Kiến trúc tổng thể & các socket sử dụng

- Hệ thống gồm 2 tiến trình chính:

a. RTSP Client:

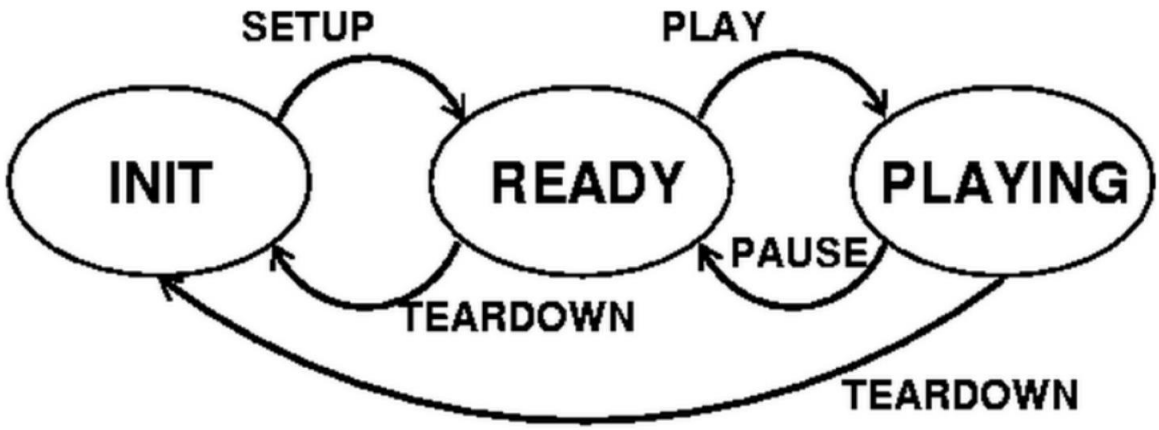
- Được khởi động qua **ClientLauncher.py**, tạo cửa sổ giao diện với 4 nút: **Setup, Play, Pause, Teardown.**

- Khi chạy, client:

- Tạo **một TCP socket** kết nối đến server trên cổng **RTSP** (ví dụ 8554). Socket này **chỉ dùng để gửi/nhận thông điệp RTSP.**

- Sau khi nhận được phản hồi SETUP 200 OK, client tạo thêm **một UDP socket** trên cổng **RTP** (ví dụ 9999) để lắng nghe các gói RTP chứa nội dung video.

- Client đồng thời duy trì **trạng thái phiên** (INIT, READY, PLAYING). Sau mỗi phản hồi của server, trạng thái được cập nhật theo sơ đồ trạng thái RTSP:



b. RTSP/RTP Server

- Module **Server.py** lắng nghe kết nối **RTSP qua TCP** từ client. Mỗi kết nối RTSP được xử lý bởi một **ServerWorker** riêng.

- Trong mỗi phiên, sau khi nhận các lệnh RTSP:

- **SETUP:** server lưu thông tin phiên (Session ID, địa chỉ & cổng RTP mà client khai báo) và chuẩn bị đối tượng **VideoStream** để đọc file video MJPEG (movie.Mjpeg hoặc MJPEG HD).
- **PLAY:** server sử dụng VideoStream để đọc lần lượt từng khung hình JPEG, đóng gói vào các gói **RTP** và gửi đều đặn tới (client_ip, client_rtp_port) qua **UDP** (mặc định khoảng 1 frame / 50 ms).
- **PAUSE:** tạm dừng vòng lặp gửi RTP nhưng vẫn giữ lại trạng thái phiên.
- **TEARDOWN:** dừng hẳn luồng gửi, đóng UDP socket và giải phóng tài nguyên phiên.

- Ở phía **client**, sau khi nhận **SETUP 200 OK**, client cũng mở một UDP socket và bind vào client_port để nhận các gói RTP từ server (chi tiết trình bày ở mục II.2).

- Như vậy, mỗi phiên stream có 2 socket chính:

Chức năng	Giao thức	Kiểu socket	Công dụng chính
Kênh điều khiển	RTSP	TCP	Trao đổi lệnh SETUP/PLAY/PAUSE/TEARDOWN, mã trạng thái
Kênh media	RTP	UDP	Gửi/nhận các gói chứa frame video

3. Định dạng video & đóng gói RTP

- File video mặc định là **movie.Mjpeg,** sử dụng **định dạng MJPEG riêng của lab**: nhiều ảnh JPEG nối liên tiếp, mỗi ảnh có **header 5 byte** chứa độ dài (số bit) của ảnh. Server dùng class **VideoStream** để đọc từng frame, không cần giải mã/encode lại. Để hỗ trợ các file **MJPEG HD (720p/1080p)** ngoài **movie.Mjpeg**, class **VideoStream** được mở rộng thêm chế độ **mjpeg**: "chương trình đọc toàn bộ dữ liệu và dò cặp byte **SOI (0xFFD8) – EOI (0xFFD9)** để tách từng frame JPEG nối tiếp. Nhờ đó server có thể stream cả file **movie.Mjpeg** chuẩn lab lẫn các file **MJPEG HD**."

- Ở phía server, class **RtpPacket** chịu trách nhiệm **đóng gói frame thành gói RTP:**

- Ghi 12 byte header: Version = 2, Padding/Extension/CC/Marker = 0 (bản cơ bản), Payload Type = 26 (MJPEG), Sequence Number = số frame, Timestamp = thời gian gửi, SSRC = ID của server.
- Sao chép dữ liệu frame (payload) vào sau header và gửi đi qua UDP.

- Ở phía client, **RtpPacket** cung cấp decode để **bóc tách header + payload**, sau đó client ghép payload thành ảnh JPEG rồi hiển thị lên GUI như một "video".

4. Phần mở rộng HD Video Streaming

- Trên nền kiến trúc trên, đồ án được mở rộng thêm:

- **Phân mảng frame (fragmentation):** nếu kích thước frame lớn hơn MTU, server cắt thành nhiều gói RTP nhỏ, sử dụng **bit Marker (M)** để đánh dấu **gói cuối** của frame. Kích thước payload tối đa mỗi gói RTP được chọn khoảng **1400 byte** (< MTU Ethernet 1500 byte) để tránh **IP fragmentation**, giảm xác suất mất gói khi stream video 720p/1080p.
- **Smooth playback with low latency:** client dùng **buffer frame** + **cơ chế timing** để phát video mượt ở 720p, đồng thời tránh trễ quá lớn.
- **Frame loss & network usage analysis:** mỗi lần người dùng **PAUSE** hoặc **TEARDOWN**, client in thống kê cho *phiên PLAY hiện tại*:
  - Dựa trên sequence number đầu tiên/cuối cùng và số frame hoàn chỉnh nhận được (sau khi qua pre-buffer) để tính **RTP Frame Loss Rate**.

- Dựa trên tổng số byte RTP nhận được và khoảng thời gian từ gói đầu tiên đến lúc PAUSE/TEARDOWN để tính **Video data rate (bytes/sec)**.
- Khi bật client-side caching, giai đoạn **pre-buffer** không được đưa vào thống kê, nên tỉ lệ mất frame phản ánh đúng phần video mà người dùng thực sự xem.

III. Kiến trúc socket & hiện thực các chức năng cơ bản

1. Mô hình client-server và các kênh truyền

- Đồ án sử dụng mô hình client–server cổ điển với **hai kênh độc lập**: kênh điều khiển RTSP (TCP) và kênh media RTP (UDP).

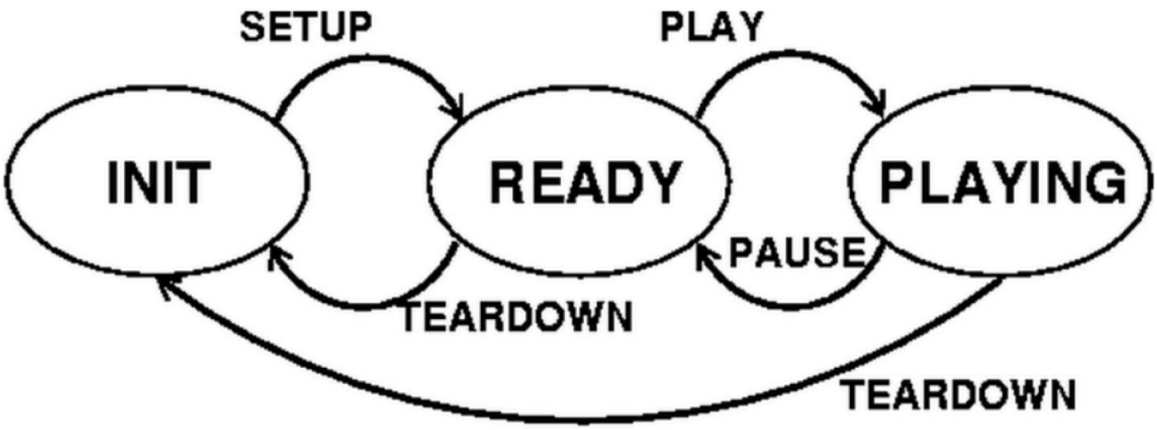
+ **Server**:

- Mở file video định dạng MJPEG (movie.Mjpeg hoặc các file HD 720p/1080p).
- Dùng VideoStream đọc lần lượt từng frame JPEG.
- Đóng gói từng frame JPEG thành các gói RTP và gửi cho client qua UDP.
- Nhận và xử lý các lệnh điều khiển RTSP (SETUP, PLAY, PAUSE, TEARDOWN).

+ **Client** (Client.py):

- Tạo socket **TCP** để gửi lệnh **RTSP** tới server (kênh điều khiển).
- Tạo socket **UDP** để nhận gói **RTP** từ server (kênh media).
- Giải mã gói RTP, hiển thị frame lên cửa sổ Tkinter và thống kê mất gói, băng thông.

- Hệ thống tuân theo **sơ đồ trạng thái RTSP**:



- Trạng thái hiện tại của client được lưu trong thuộc tính **self.state** và chỉ cho phép gửi những lệnh hợp lệ tương ứng với từng state.

2. Kiến trúc phía client (Client.py)

2.1. Các socket và biến quan trọng

- Trong lớp Client, các socket chính:

+ **self.rtspSocket** – socket **TCP** dùng cho **RTSP**:

- Được tạo trong **connectToServer()** ngay khi khởi tạo client.
- Kết nối tới **(serverAddr, serverPort)** và được giữ nguyên cho **toàn bộ phiên RTSP.**
- Dùng để **gửi** các request SETUP / PLAY / PAUSE / TEARDOWN và **nhận** RTSP reply (mã trạng thái 200 OK, Session ID, v.v.).

+ **self.rtpSocket** – socket **UDP** dùng cho **RTP**;

- Được tạo trong **openRtpPort() sau khi nhận SETUP 200 OK** (khi state chuyển sang READY).
- Được **bind** vào **self.rtpPort** mà client khai báo trong request:

```
f"Transport: RTP/UDP; client_port= {self.rtpPort}"
```

- Được cấu hình **settimeout(0.5)** để recv() không bị block vô thời hạn; mọi gói RTP sẽ được đọc trong **thread listenRtp().**
- Socket này chỉ được đóng thật sự khi **TEARDOWN** hoàn tất (dựa trên cờ **self.teardownAcked**).

- Các biến điều khiển phiên RTSP:

+ self.state: INIT, READY, PLAYING.

+ **self.rtspSeq**: số thứ tự CSeq của RTSP, **tăng dần cho mỗi request**; dùng để đối chiếu với sequence trong RTSP reply.

+ **self.sessionId**: Session ID do server cấp sau lệnh SETUP đầu tiên; các request sau (PLAY/PAUSE/TEARDOWN) phải gửi kèm đúng Session ID này.

+ **self.requestSent**: lưu lại request RTSP cuối cùng, phục vụ xử lý reply.

+ **self.teardownAcked**: cờ (0/1) báo đã nhận RTSP reply 200 OK cho TEARDOWN; thread listenRtp() dựa vào cờ này để chủ động shutdown() và close() rtpSocket.

+ **self.playEvent**: một threading.Event dùng để **ra hiệu dừng** vòng lặp trong listenRtp() khi người dùng PAUSE hoặc TEARDOWN. Khi playEvent.set() được gọi, thread nhận RTP sẽ thoát ra an toàn.

- Các biến hỗ trợ giải mã & hiển thị frame:

+ **self.frameNbr**: số thứ tự frame RTP **cao nhất** đã xử lý, dùng để nhận diện khi bắt đầu một frame mới (seqNum tăng) và bỏ qua các gói đến trễ nếu seqNum < frameNbr.

+ **self.frameBuffer**: một bytearray dùng làm **buffer tạm cho HD Video Streaming**, để ghép nhiều fragment (gói RTP có cùng seqNum) thành **một frame JPEG hoàn chỉnh** trước khi ghi ra file/cache.

+ **self.targetInterval**: khoảng thời gian mong muốn giữa hai frame (mặc định 0.05 s ≈ 20 fps), phục vụ cơ chế **Smooth HD playback**.

+ **self.lastFrameTime**: thời điểm lần cuối cùng frame được hiển thị; dùng chung với targetInterval để chỉnh sleep() cho FPS ổn định, tránh phát quá nhanh.

- Các biến thống kê frame loss & băng thông:

+ **self.firstSeq**: sequence number **đầu tiên** nhìn thấy (sau khi bắt đầu tính thống kê).

+ **self.maxSeq**: sequence number **lớn nhất** đã nhận được.

+ **self.framesReceived**: số frame **hoàn chỉnh** (nhận đủ fragment, marker=1) sau giai đoạn pre-buffer; dùng để tính **RTP Frame Loss Rate**.

+ **self.frameDisplayed:** số frame thật sự đã hiển thị lên GUI (dùng để debug/quan sát thêm).

+ **self.totalBytes**: tổng số byte RTP đã nhận, dùng để tính **Video data rate (bytes/sec), Tỉ lệ mất frame RTP Frame Loss Rate**

+ **self.startTime**: thời điểm nhận packet đầu tiên dùng cho thống kê (được reset mỗi lần PLAY).

- Hàm reportStats() sử dụng các biến này để in ra:

+ Tỉ lệ mất frame RTP Frame Loss Rate = lostFrames / totalFrames.

+ Tốc độ dữ liệu Video data rate = totalBytes / duration.

- Các biến cho Client-Side Caching (frame buffer phía client):

+ **self.enableCache**: bật/tắt toàn bộ tính năng client-side caching.

+ **self.cachePreload:** số frame **N** sẽ được **pre-buffer** trước khi bắt đầu hiển thị (trong code hiện tại N = 10).

+ **self.cacheMaxSize**: kích thước tối đa của buffer phía client (số frame tối đa lưu trong frameCache).

+ **self.frameCache**: một deque (hàng đợi FIFO) chứa các frame JPEG đã hoàn chỉnh, được dùng để tích lũy N frame đầu trong giai đoạn pre-buffer và sau đó làm buffer trượt trong giai đoạn "Playing from buffer".

+ **self.cachePrebuffering**: cờ True/False:

- True khi đang trong giai đoạn **Prebuffering N frame** (chỉ nhận và tích vào frameCache, chưa show).
- Khi đủ N frame, đặt False → bắt đầu **PLAY từ buffer**, mỗi frame mới đến sẽ đẩy vào queue, đồng thời pop frame cũ nhất để hiển thị.

2.2. Xử lý các nút lệnh RTSP

- Giao diện Tkinter của client cung cấp bốn nút **Setup, Play, Pause, Teardown**.
Mỗi nút tương ứng với một lệnh RTSP được gửi tới server và làm thay đổi trạng thái phiên (INIT → READY → PLAYING).

a) Nút SETUP:

- Khi người dùng bấm **Setup** và client đang ở trạng thái INIT, hàm **setupMovie()** gửi một yêu cầu **RTSP SETUP** đến server.

- Yêu cầu này khai báo tên file video, số thứ tự CSeq và cổng RTP mà client sẽ dùng để nhận dữ liệu (client_port = rtpPort).

- Sau khi nhận phản hồi **200 OK** với Session ID hợp lệ, client:

- Cập nhật sessionId và chuyển trạng thái sang **READY**.
- Gọi **openRtpPort()** để mở socket UDP **rtpSocket** và bind vào **rtpPort**, chuẩn bị cho việc nhận các gói RTP từ server.

```
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 = (
        f"SETUP {self.fileName} RTSP/1.0\n"
        f"CSeq: {self.rtspSeq}\n"
        f"Transport: RTP/UDP; client_port= {self.rtpPort}"
    )

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

```
# 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()
```

```
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)
```

b) Nút PLAY:

- Khi ở trạng thái **READY**, người dùng nhấn **Play** sẽ kích hoạt hàm **playMovie().**

- Client tạo một thread mới chạy **listenRtp()** để lắng nghe và xử lý các gói **RTP** trên **rtpSocket**, sau đó gửi yêu cầu **RTSP PLAY** kèm **Session ID** tới server.

- Khi server trả về **PLAY 200 OK**, client:

- Chuyển trạng thái sang **PLAYING**.
- Reset các biến thống kê (firstSeq, maxSeq, framesReceived, totalBytes, startTime, …) để đo **frame loss rate** và **video data rate** cho phiên phát này.
- Nếu tính năng **Client-Side Caching** được bật, buffer frameCache được xóa và bật cờ cachePrebuffering để chuẩn bị giai đoạn **pre-buffer N frame** trước khi hiển thị.

```
# 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 = (
        f"PLAY {self.fileName} RTSP/1.0\n"
        f"CSeq: {self.rtspSeq}\n"
        f"Session: {self.sessionId}"
    )

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

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

    # Reset thống kê cho mỗi lần PLAY (mỗi session xem mới) ->
    # để mỗi lần bấm PLAY sẽ có 1 bộ stats riêng, không bị dính với lần PLAY trước
    # Khi cache bật, thống kê sẽ bỏ qua hoàn toàn giai đoạn prebuffer (vì lúc đó cach
    # startTime chỉ bắt đầu lại khi thật sự đếm loss (sau prebuffer)
    self.firstSeq = None
    self.maxSeq = 0
    self.frameDisplayed = 0
    self.totalBytes = 0
    self.startTime = None
    self.lastFrameTime = None
    self.framesReceived = 0

    # Client-side caching: reset buffer + bật pre-buffer
    if self.enableCache:
        self.frameCache.clear()
        self.cachePrebuffering = True
```

c) Nút PAUSE

- Khi đang ở trạng thái **PLAYING**, nút **Pause** gửi yêu cầu **RTSP PAUSE** đến server.

- Sau khi nhận **PAUSE 200 OK**, client:

  - Đưa trạng thái về **READY**.

- Gọi **reportStats()** để in ra **tỉ lệ mất frame** và **tốc độ dữ liệu video** tính được từ đầu phiên PLAY đến thời điểm tạm dừng.
- Đặt cờ **playEvent** để thread **listenRtp()** thoát khỏi vòng lặp nhận RTP và dừng lại an toàn.

- Lưu ý: socket RTP **không bị đóng** ở bước này; nếu người dùng nhấn Play lại, client chỉ cần tạo thread listenRtp() mới và tiếp tục nhận dữ liệu trên cùng socket.

```
# 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 = (
        f"PAUSE {self.fileName} RTSP/1.0\n"
        f"CSeq: {self.rtspSeq}\n"
        f"Session: {self.sessionId}"
    )

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

```
elif self.requestSent == self.PAUSE:
    self.state = self.READY

    # In thống kê ngay khi server xác nhận PAUSE
    self.reportStats()

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

d) Nút TEARDOWN

- Nút **Teardown** dùng để kết thúc hoàn toàn phiên stream (có thể bấm từ READY hoặc PLAYING).

- Khi được nhấn, **hàm exitClient():**

- Đặt **playEvent** (nếu tồn tại) để dừng thread nhận RTP.

- Gửi yêu cầu **RTSP TEARDOWN** đến server.
- Xóa file ảnh tạm cache-<sessionId>.jpg nếu còn tồn tại và đóng cửa sổ GUI.

- Khi server trả về **TEARDOWN 200 OK**, client:

- Chuyển trạng thái về **INIT** và đặt cờ teardownAcked = 1.
- Gọi **reportStats()** lần cuối để thống kê tổng thể.
- Nếu đang bật **caching**, xóa nội dung **frameCache** và tắt cờ **cachePrebuffering**.

- Trong thread listenRtp(), khi thấy teardownAcked == 1, client sẽ chủ động **shutdown và close** socket UDP rtpSocket, hoàn tất việc giải phóng tài nguyên mạng.

```
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 = (
        f"TEARDOWN {self.fileName} RTSP/1.0\n"
        f"CSeq: {self.rtspSeq}\n"
        f"Session: {self.sessionId}"
    )

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

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

    # Flag the teardownAcked to close the socket.
    self.teardownAcked = 1

    # In thống kê ngay khi server xác nhận TEARDOWN
    self.reportStats()

    # Client-side caching: dọn buffer
    if self.enableCache:
        self.frameCache.clear()
        self.cachePrebuffering = False
```

3. Module gói RTP – RtpPacket.py

3.1 Cấu trúc gói RTP

- Header RTP gồm 12 bytes đầu:

+ Byte 0: 2 bit **Version (V)** – luôn đặt 2, 1 bit **Padding (P)** – lab đặt 0, 1 bit **Extension (X)** – lab đặt 0, 4 bit **CC** – số CSRC, lab đặt 0.

+ Byte 1: 1 bit **Marker (M)** – trong chế độ **cơ bản** đặt **0**, **khi mở rộng HD được dùng để đánh dấu gói cuối cùng của một frame (server đặt M = 1)**, 7 bit Payload Type (PT) – lab dùng 26 cho MJPEG.

+ Byte 2 – 3: **Sequence Number** – tăng 1 cho mỗi frame/packet.

+ Byte 4 – 7: **Timestamp**.

+ Byte 8 – 11: **SSRC** – lab đặt 0.

- Trong lớp **RtpPacket**, mỗi gói RTP được biểu diễn bằng hai trường **self.header** (mảng 12 byte header) và **self.payload** (dữ liệu JPEG). Hai trường này được gán tương ứng trong các hàm **encode () (khi server tạo gói)** và **decode () (khi client nhận gói).**

3.2. Hàm encode() – đóng gói

- Hàm encode() nhận các tham số header (version, padding, extension, CC, sequence number, marker, payload type, SSRC) và payload là dữ liệu JPEG:

- Tạo một mảng **header** 12 byte và gán từng trường theo đúng layout của chuẩn RTP (shift & mask từng byte).
- **timestamp** được lấy từ thời gian hiện tại **(int (time ()))**, đảm bảo tăng dần theo thời gian gửi.
- **Bit Marker (M)** được truyền từ tham số: trong chế độ **cơ bản M = 0**; với **HD streaming** khi phân mảnh frame, server đặt **M = 1 cho gói cuối cùng của mỗi frame.**
- Sau khi gán xong header, hàm lưu **self.header = header** và **self.payload = payload**; **getPacket()** chỉ đơn giản trả về header + payload để gửi qua **UDP**.

```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()) # RTP timestamp: dùng thời gian hiện tại (giây, tăng dần theo thời gian)
    header = bytearray(HEADER_SIZE) # 12-byte RTP header (mutable để gán từng byte)
    #--------------
    # TO COMPLETE
    #--------------
    # Fill the header bytearray with RTP header fields

    # Byte 0: V(2) | P(1) | X(1) | CC(4)
    header[0] = ((version & 0x03) << 6) | ((padding & 0x01) << 5) | ((extension & 0x01) << 4) | (cc & 0x0F)
    # Byte 1: M(1) | PT(7)
    header[1] = ((marker & 0x01) << 7) | (pt & 0x7F)
    # Sequence number (16 bits)
    header[2] = (seqnum >> 8) & 0xFF
    header[3] = seqnum & 0xFF
    # Timestamp (32 bits)
    header[4] = (timestamp >> 24) & 0xFF
    header[5] = (timestamp >> 16) & 0xFF
    header[6] = (timestamp >> 8) & 0xFF
    header[7] = timestamp & 0xFF
    # SSRC (32 bits) from argument
    header[8] = (ssrc >> 24) & 0xFF
    header[9] = (ssrc >> 16) & 0xFF
    header[10] = (ssrc >> 8) & 0xFF
    header[11] = ssrc & 0xFF

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

3.3. Hàm decode() và các getter

- Ở phía client, khi nhận một gói RTP, code tạo đối tượng:

```python
if data: # nếu có dữ liệu tiếp tục giải mã
    rtpPacket = RtpPacket()
    rtpPacket.decode(data)
```

- decode() tách 12 byte đầu làm header, phần còn lại là payload, sau đó các hàm:

- **version(), seqNum(), timestamp (), payloadType()** đọc từng trường bằng bit-mask & shift.
- **getPayload()** trả về dữ liệu JPEG.
- **marker ()** (phần mở rộng HD) cho biết gói cuối cùng của một frame khi có phân mảnh.

- Nhờ mô-đun **RtpPacket**, việc **packetization** và **depacketization** được đóng gói gọn trong một lớp.

IV. Tổng kết kiến trúc socket & nhận xét

- Hệ thống dùng 2 kênh: **RTSP/TCP** để **điều khiển phiên**, **RTP/UDP** để **truyền media**.

- Phía client: một **rtspSocket (TCP)** duy trì suốt phiên, một **rtpSocket (UDP)** mở sau **SETUP** và đóng khi **TEARDOWN**; dùng **thread** + **event** để quản lý.

- Mở rộng **HD streaming**: thêm fragmentation (**Marker bit**), **buffer + timing** và **client-side caching** để giảm **jitter**; thống kê **frame loss** & **data rate** để đánh giá mạng.