Video streaming with RTSP and RTP

I. Giới thiệu:

1. Mục tiêu:

Xây dựng và triển khai một hệ thống video streaming đơn giản, mô phỏng kiến trúc thực tế trên Internet. Hệ thống bao gồm: server phát video và client nhận. Trong đó, hai giao thức chính được sử dụng bao gồm:

- Kênh điều khiển (Control Channel): thiết lập phiên (session) để stream video và kiểm soát các thao tác của client (SETUP, PLAY, PAUSE, TEARDOWN).
  Giao thức chính là *Real-time Streaming Protocol (RTSP)* sử dụng *TCP*
- Luồng dữ liệu (Media Stream): truyền dữ liệu là video thực tế thông qua việc đóng gói (packetization) bằng phương thức *Real-time Transfer Protocol (RTP)* sử dụng *UDP*

2. Nhiệm vụ:

Nhiệm vụ trọng tâm được phân thành 3 phần chính:

- Giao thức cơ bản: Hoàn thiện triển khai giao thức RTSP phía Client và đóng gói RTP ở phía Server thông qua việc hoàn chỉnh mã nguồn còn thiếu.
- HD Video Streaming: Mở rộng hệ thống để hỗ trợ stream video với độ phân giải 720p hoặc 1080p thông qua việc triển khai phân mảnh (fragmentation) cho các khung hình vượt quá MTU (Maximum Transmission Unit - kích thước tối đa của 1 packet) nhằm tối ưu hóa hiệu suất truyền tải.
- Client-side Caching: Thiết lập bộ đệm khung hình ở phía Client để giảm hiện tượng jitter (chậm trễ trong quá trình truyền và nhận dữ liệu) bằng cách tải trước một số khung hình.

II. Triển khai giao thức cơ bản:

Hệ thống video streaming sử dụng kiến trúc phân tách chức năng. Trong đó, hai giao thức được sử dụng cho hai chức năng riêng biệt:

- RTSP trên TCP: Sử dụng cho kênh điều khiển vì việc thiết lập kết nối cần có mức độ tin cậy cao => sử dụng TCP vì tính chất tương tự.
- RTP trên UDP: Sử dụng cho luồng dữ liệu vì việc phát đi dữ liệu cho client có thể xem không nhất thiết phải tin cậy mà phải đảm bảo tốc độ truyền tối đa.

1. Kênh điều khiển (Control Channel)

Kênh điều khiển sẽ thiết lập phiên (session) và quản lý các thao tác mà client yêu cầu.

Giao thức: Real-time Streaming Protocol (RTSP)

Giao vận: Transmission Control Protocol (TCP) để đảm bảo độ tin cậy trong quá trình truyền tải thông điệp quan trọng.

Cổng mặc định: 554

1.1 Các lệnh RTSP:

Client sẽ gửi lệnh văn bản đến server thông qua cổng TCP:

- **SETUP**: C yêu cầu S chuẩn bị một luồng video cụ thể và cổng UDP cụ thể để nhận dữ liệu.
- **PLAY**: C ra lệnh S bắt đầu truyền dữ liệu video qua RTP/UDP.
- **PAUSE**: C ra lệnh S tạm dừng việc gửi dữ liệu video nhưng vẫn duy trì trạng thái **READY**
- **TEARDOWN**: C ra lệnh S kết thúc phiên làm việc và đóng kết nối với S.

1.2 Triển khai RTSP bên Client:

Triển khai bằng cách Client gửi request qua socket TCP đã được thiết lập sẵn. Client phải theo dõi trạng thái phiên (session state) và số thứ tự lệnh (CSeq).

- CSeq (Client-sequence): Là biến đếm được tăng lên 1 sau mỗi lần 1 request được gửi đi. Header `Cseq: {self.rtspSeq}` được chèn vào tất cả các request. Nhằm đảm bảo sự tin cậy và thứ tự cho các lệnh điều khiển.
- Session ID: Nhằm duy trì trạng thái của phiên làm việc. Nếu lệnh **SETUP** trả về phản hồi **200 OK**, Client trích xuất Session ID từ header `Session: {self.sessionId}` của Server và sau đó được chèn vào tất cả các request tiếp theo.

Server quản lý trạng thái của Client kể từ phản hồi thành công:

Lệnh	Trạng thái chuyển đổi	Hành động của Client	Header
SETUP	INIT $\to$ READY	Tạo socket UDP để nhận dữ liệu RTP và thiết lập timeout 0.5s. Gửi Request và chờ Session ID.	Transport: Chỉ định cổng UDP của Client (client_port= <RTP_port>).
PLAY	READY $\to$ PLAYING	Bắt đầu nhận dữ liệu RTP (Video) qua UDP.	Session: Chứa Session ID đã nhận được từ SETUP. Không chèn header Transport.
PAUSE	PLAYING $\to$ READY	Dừng nhận dữ liệu RTP.	Session: Chứa Session ID. Không chèn header Transport.
TEARDOWN	READY/PLAYING $\to$ INIT	Đóng socket TCP RTSP và socket UDP RTP.	Session: Chứa Session ID22. Không chèn header Transport23.

1.3 Xử lý các yêu cầu từ client đến server:

Các trạng thái khi stream video và các lệnh từ phía Client sẽ được mô tả thông qua giá trị của các biến trong class `Client`.

Các request được xây dựng bằng cách nối chuỗi các header bắt buộc thông qua hàm `sendRtspRequest` và sau đó sẽ được gửi đến server thông qua socket đã thiết lập.

```
    class Client:
        INIT = 0
        READY = 1
```

```
        PLAYING = 2
        state = INIT

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

        # khởi tạo CSeq và SessionId bằng 0
        def __init__(self, master, serveraddr, serverport, rtpport, filename):
            self.rtspSeq = 0
            self.sessionId = 0
            self.requestSent = -1 # lưu lại yêu cầu được gửi để xử lý trạng thái
            ...
```

```
    # gửi yêu cầu đến server
    def sendRtspRequest(self, method):
        self.rtspSeq += 1

        request = f"{method} {self.fileName} RTSP/1.0\r\n"
        request += f"CSeq: {self.rtspSeq}\r\n"

        if (method != 'SETUP' and self.state != 'INIT'):
            request += f"Session: {self.sessionId}\r\n"

        if (method == 'SETUP'):
            request += f"Transport: RTP/UDP; client_port={self.rtpPort}\r\n"

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

```
    # xử lý phản hồi từ server
    def parseRtspReply(self, data):
        seqNum <- xử lý data thông qua các phương thức xử lý chuỗi
        # chỉ xử lý nếu seqnum của server trùng với client
        if (seqNum == self.rtspSeq):
            # tạo mới
            if (self.sessionId == 0):
                self.sessionId = session
                return

            if (self.requestSent == 'SETUP'):
                self.state = 'READY'
                self.openRtpPort() # mở cổng Rtp để nhận dữ liệu từ server

            elif (self.requestSent == 'PAUSE'):
                self.state = 'READY'
                self.playEvent.set() # luồng chạy video đã có, tạo luồng mới để
```

```
tiếp tục

            elif (self.requestSent == 'TEARDOWN'):
                self.state = 'INIT'
                self.teardownAcked = 1 # flag để đóng socket
```

1.4 Tạo socket RTP nhận dữ liệu từ server:

Tạo socket UDP thông qua các phương thức được hỗ trợ từ thư viện `socket` của python.

```
    # mở cổng rtp để nhận dữ liệu
    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:
            # gửi thông báo nếu không bind được socket
```

Thiết lập timeout 0.5s cho RTP socket của client để chuyển socket từ chế độ chặn sang chế độ không chặn trong vòng 0.5s.

Nếu không có timeout, lệnh nhận dữ liệu (`recvfrom()`) sẽ chặn luồng của client vô thời hạn nếu không có gói tin nào đến.

Đặt timeout để đảm bảo sau mỗi 0.5s, luồng của client sẽ giải phóng khỏi lệnh `recvfrom()` để kiểm tra một số yếu tố khác như:

- Kiểm tra trạng thái: kiểm tra xem người dùng có nhấn nút PAUSE hoặc TEARDOWN không để thoát khỏi vòng lặp nhận dữ liệu.
- Xử lý các gói tin bị mất: cho phép client bỏ qua các gói tin bị mất và tiếp tục vòng lặp chờ đợi cho gói tin tiếp theo.

Cuối cùng, gán socket UDP với địa chỉ IP và hiệu số cổng cụ thể được cung cấp bởi client `client_port={self.rtpPort}`.

2. Kênh dữ liệu (Data Channel)

Kênh dữ liệu sẽ chịu trách nhiệm trong việc phân phối và truyền tải dữ liệu video đến bên client từ server.

Giao thức: Real-time Transport Protocol (RTP)

Giao vận: User Datagram Protocol (UDP) để đảm bảo quá trình truyền dữ liệu ở tốc độ tối đa và độ trễ thấp bởi nhu cầu thiết yếu của quá trình video streaming.

2.1 Cơ chế hoạt động:

Phía server (RTP Packetization):

- Server (S) đọc khung hình: S đọc tuần tự từng khung hình (trong file .Mjpeg thì mỗi file jpeg là 1 frame hoàn chỉnh)
- Đóng gói RTP (RTP packetization): S tạo 1 gói tin RTP bằng cách ghép 1 khung hình vừa đọc (có thể gọi là Payload) với 1 file header RTP cố định **12 byte**.
- Truyền tải UDP: S gửi gói RTP này đến cổng UDP của client.

Phía client (RTP De-packetization):

- Client (C) nhận datagram: C nhận datagram UDP chứa gói tin RTP.
- Tách gói tin: C tiến hành phân tích gói tin với 12 byte đầu là header RTP và phần payload.
- Tái tạo và phát: C sử dụng `seqnum` để sắp xếp lại gói tin nếu nó không đến đúng thứ tự và sử dụng `timestamp` để đồng bộ hóa và điều chỉnh thời điểm phát khung hình, quản lý độ trễ (jitter) đảm bảo phát video đồng đều.

2.2 Cấu trúc gói tin RTP:

Mỗi gói tin RTP sẽ bao gồm 2 phần: header và payload. Trong đó, header gồm 12 byte cố định và payload là dữ liệu của mỗi khung hình.

- Header:

Field	Độ dài (bits)	Vai trò
V (version)	2	Cố định bằng 2 (đối với RTP)
PT (playload type)	7	Cố định bằng 26 (tương thích với định dạng .Mjpeg)
Sequence number	16	Tăng lên 1 sau mỗi gói tin, giúp client sắp xếp thứ tự gói tin và phát hiện mất gói
Timestamp	32	Cập nhật theo thời gian, giúp client đồng bộ hóa việc phát video
SSRC	32	Trường 32-bit xác định Server
Marker bit (M)	1	Đặt là 0 trong gói tin RTP. Đặc biệt quan trọng trong các gói tin được **phân mảnh** (fragmentation) của HD streaming để đánh dấu ranh giới khung hình

- Payload: chứa dữ liệu thực tế mà người dùng cần.

2.3 Triển khai đóng gói RTP bên Server:

Việc triển khai này nằm trong hàm `encode` có nhiệm vụ đóng gói một khung hình video thành 1 gói tin RTP rồi gửi qua UDP mỗi 50 mili giây.

Header của lớp RtpPacket có kiểu `bytearray` (mảng các byte) nên cần phải thiết lập từng byte một.

Sử dụng các toán tử thao tác bit (bit manipulation)

- Thiết lập bit n trong biến `mybyte`:
  `mybyte = mybyte | 1 << (7 - n)`
- Thiết lập bit n và n + 1 thành giá trị `foo` trong biến `mybyte`:
  `mybyte = mybyte | foo << (7 - n)`
- Sao chép số nguyên 16-bit vào 2 byte b1 và b2:
  `b1 = (foo >> 8) & 0xFF`
  `b2 = foo & 0xFF`

```
    def encode(self, version, padding, extension, cc, seqnum, marker, pt, ssrc,
payload):
        timestamp = int(time())

        self.header[0] = (version << 6) | (padding << 5) | (extension << 4) | cc

        self.header[1] = (marker << 7) | pt

        # sequence number
        self.header[2] = (seqnum >> 8) & 0xFF
        self.header[3] = seqnum & 0xFF

        # timestamp
        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) & 0xFF
        self.header[9] = (ssrc >> 16) & 0xFF
        self.header[10] = (ssrc >> 8) & 0xFF
        self.header[11] = ssrc & 0xFF

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

3. Chạy mã nguồn:

Sử dụng terminal trên command prompt của thiết bị hoặc tích hợp sẵn trong Visual Studio Code. Di chuyển đến directory chứa file mã nguồn.

- Bên server: `py .\Server.py <server_port>`
- Bên client: `py .\ClientLauncher.py <server_host> <server_port> <rtp_port> <video_file>`

Đối với client, trước khi thiết lập kết nối với server cần kích hoạt môi trường ảo của python để đảm bảo các thư viện trong quá trình video streaming.

```
<path to your code directory>\.venv\Scripts\activate
```