VNU-HCM
UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

PROJECT: VIDEO STREAMING WITH RTSP & RTP

COMPUTER NETWORKING

Lecturer:
MSc. Le Ha Minh

Students:
24127424 - Trần Anh Khoa
20127115 - Lâm Quốc Bảo

HO CHI MINH CITY, 2025

Mục lục

1 Part 1: System Architecture

1.1 Overview

The system implements a Video Streaming application based on the Client-Server model. It utilizes two distinct network protocols to optimize performance:

- **RTSP (Real-Time Streaming Protocol):** Runs over TCP (Port 1025) to manage the session states (INIT, READY, PLAYING). TCP ensures that control commands like Play/Pause are delivered reliably without loss.
- **RTP (Real-time Transport Protocol):** Runs over UDP (Port 5000) to transport the actual video frames. UDP is chosen for its low latency, which is critical for real-time media.

2 Part 2: Implementation Details

This chapter details the core algorithms and logic used to build the system, including the bit-level construction of headers and advanced flow control mechanisms.

2.1 1. RTP Packet Construction (Server Side)

The server must encapsulate video data into RTP packets before sending them over UDP. We implemented the standard RTP header (RFC 1889) in `RtpPacket.py`.

Bit Manipulation Logic: Since Python does not support C-style structures, we used bitwise operators to pack fields into a 12-byte header:

- **Byte 0:** Contains Version ($V = 2$), Padding ($P = 0$), Extension ($X = 0$), and CSRC Count ($CC = 0$). We use left shifts (`<<`) to place bits in the correct positions and OR (`|`) to combine them.
- **Byte 1:** Contains the **Marker (M)** bit and Payload Type ($PT = 26$ for MJPEG). The Marker bit is set to 1 only for the last packet of a frame.
- **Bytes 2-3 (Sequence Number):** A 16-bit integer split into two 8-bit values using masking (`& 0xFF`).

```python
def encode(self, version, padding, extension, cc, seqnum, marker, pt,
    ssrc, payload):
    """Encode the RTP packet with header fields and payload."""
    timestamp = int(time())
    header = bytearray(HEADER_SIZE) # HEADER_SIZE = 12

    # Byte 0: V(2) | P(1) | X(1) | CC(4)
    header[0] = (version << 6) & 0xC0
    header[0] = header[0] | ((padding << 5) & 0x20)
    header[0] = header[0] | ((extension << 4) & 0x10)
    header[0] = header[0] | (cc & 0x0F)

    # Byte 1: M(1) | PT(7)
    header[1] = (marker << 7) & 0x80
    header[1] = header[1] | (pt & 0x7F)

    # Byte 2-3: Sequence Number (16 bits)
    header[2] = (seqnum >> 8) & 0xFF
    header[3] = seqnum & 0xFF

    # Byte 4-7: Timestamp (32 bits)
    header[4] = (timestamp >> 24) & 0xFF
    header[5] = (timestamp >> 16) & 0xFF
    header[6] = (timestamp >> 8) & 0xFF
    header[7] = timestamp & 0xFF

    # Byte 8-11: SSRC (32 bits)
    header[8] = (ssrc >> 24) & 0xFF
    header[9] = (ssrc >> 16) & 0xFF
    header[10] = (ssrc >> 8) & 0xFF
    header[11] = ssrc & 0xFF

    self.header = header
    self.payload = payload
```

Listing 1: Full implementation of RTP Header Encoding in RtpPacket.py

2.2 2. RTSP State Machine (Client Side)

The client maintains a state machine to ensure valid transitions. The `parseRtspReply` function in `Client.py` validates server responses before changing state.

Logic: The function checks if the `Session ID` matches and if the status code is `200 OK`.

- **SETUP → READY:** Opens the RTP socket binding to the specified port.
- **PLAY → PLAYING:** Starts the receiving thread (`receiveRtp`) and display thread (`consumeBuffer`).
- **PAUSE → READY:** Sets the event to pause the display loop.
- **TEARDOWN → INIT:** Closes sockets and resets flags.

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

    # Check Sequence Number
    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
                    # Start threads if not already running
                    if not hasattr(self, 'rtp_thread'):
                        self.rtp_thread = threading.Thread(target=self.receiveRtp)
                        self.rtp_thread.start()
                        self.display_thread = threading.Thread(target=self.consumeBuffer)
                        self.display_thread.start()
                        self.isBuffering = True
                elif self.requestSent == self.PAUSE:
                    self.state = self.READY
                    self.playEvent.set()
                elif self.requestSent == self.TEARDOWN:
                    self.state = self.INIT
                    self.teardownAcked = 1
```

Listing 2: State Transition Logic in Client.py

2.3 3. Advanced Feature: Fragmentation for HD Video

The Maximum Transmission Unit (MTU) for Ethernet is typically 1500 bytes. HD video frames are much larger (e.g., 20KB - 50KB). Sending them as a single packet causes IP fragmentation, which is unreliable for UDP. We implemented Application-Layer Fragmentation in `ServerWorker.py`.

Algorithm: The `sendRtp` function splits the frame data into chunks of 1400 bytes.

- **Loop:** We iterate through the data array using a `while` loop.
- **Marker Bit:** For every chunk, we check if it is the last one.
  - If `is_last_chunk == True`, we set `marker = 1`.
  - Otherwise, `marker = 0`.
- This allows the client to know when to reassemble the frame.

```python
def sendRtp(self):
    """Send RTP packets over UDP with Fragmentation."""
    MAX_PAYLOAD_SIZE = 1400 # Safe payload size < MTU

    while True:
        self.clientInfo['event'].wait(0.04)
        if self.clientInfo['event'].isSet():
            break

        data = self.clientInfo['videoStream'].nextFrame()
        if data:
            frameNumber = self.clientInfo['videoStream'].frameNbr()
            address = self.clientInfo['rtspSocket'][1][0]
            port = int(self.clientInfo['rtpPort'])

            # --- FRAGMENTATION ALGORITHM ---
            bytes_sent = 0
            total_len = len(data)

            while bytes_sent < total_len:
                # Determine chunk size
                chunk_size = min(MAX_PAYLOAD_SIZE, total_len - bytes_sent)
                payload = data[bytes_sent : bytes_sent + chunk_size]
                bytes_sent += chunk_size

                # Set Marker=1 only for the final fragment
                is_last_chunk = (bytes_sent >= total_len)
                marker = 1 if is_last_chunk else 0

                self.clientInfo['rtpSeqNum'] += 1

                try:
                    packet = self.makeRtp(payload, self.clientInfo['rtpSeqNum'], marker)
                    self.clientInfo['rtpSocket'].sendto(packet, (address, port))
                except Exception as e:
                    print("Connection Error: ", e)
```

Listing 3: Fragmentation Logic in ServerWorker.py

2.4 4. Advanced Feature: Client-Side Buffering

Network jitter causes packets to arrive at irregular intervals. To ensure smooth playback (30 FPS), we implemented a Jitter Buffer using a thread-safe Queue.

Mechanism: The `consumeBuffer` function in `Client.py` acts as the consumer.

- **Pre-buffering:** When Play starts, the client enters "Buffering" mode. It waits until the queue size reaches `MIN_BUFFER_SIZE` (e.g., 20 frames).
- **Underrun Protection:** If the queue becomes empty during playback, the client automatically pauses display and shows "Buffering..." again until more frames arrive.

```python
def consumeBuffer(self):
    """Consume frames from buffer and display."""
    while True:
        if self.exitEvent.is_set(): break

        # Check Buffer Level
        if self.frameBuffer.qsize() < MIN_BUFFER_SIZE and self.isBuffering:
            print("Buffering...") # Wait for buffer to fill
            time.sleep(0.1)
            continue
        else:
            self.isBuffering = False # Buffer is healthy

        if not self.frameBuffer.empty():
            # Get frame and display
            frameData = self.frameBuffer.get()
            self.updateMovie(self.writeFrame(frameData))

            self.stat_framesDisplayed += 1
            # Sleep to maintain frame rate
            time.sleep(frame_delay)
        else:
            # Buffer empty! Re-enter buffering mode
            print("Buffer empty! Re-buffering...")
            self.isBuffering = True
```

Listing 4: Buffering Logic in Client.py

3 Part 3: Demonstration Results

This section demonstrates the complete workflow of the application, verifying all functionalities from connection establishment to session teardown.

3.1 State 1: System Initialization

Action: Start the Server and Client from the command line.

- Server terminal shows listening status on port 1025.
- Client GUI launches successfully in the INIT state (buttons enabled, screen blank).

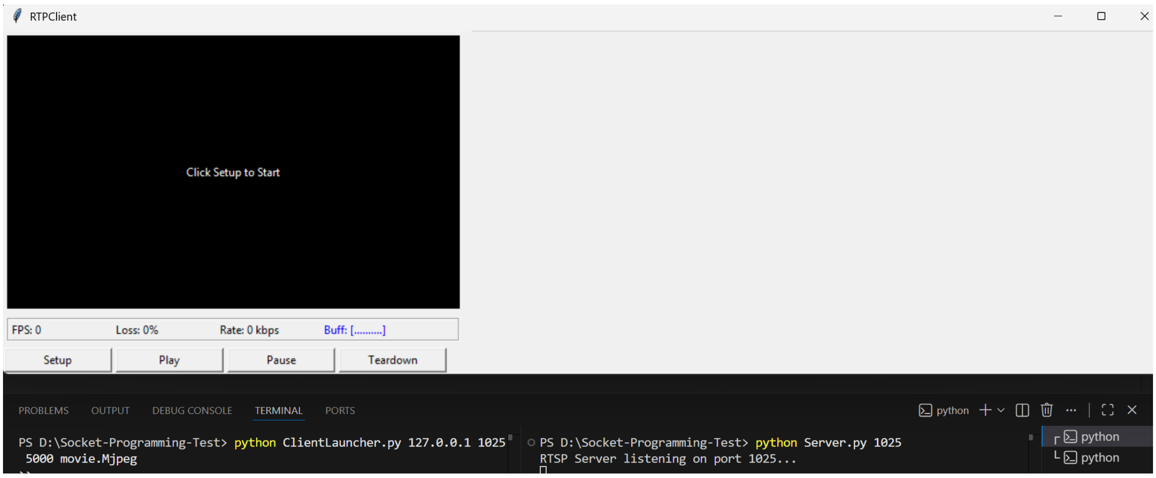

Figure 1: System Initialization: Server Terminal and Client GUI

3.2 State 2: Setup Phase

Action: User clicks the **Setup** button.

- Client terminal logs:

```
Data sent:
SETUP movie.Mjpeg RTSP/1.0
CSeq: 1
Transport: RTP/UDP; client_port= 5000
```

- Server terminal logs: processing SETUP
- Client receives 200 OK and logs: `Transition: INIT -> READY`.

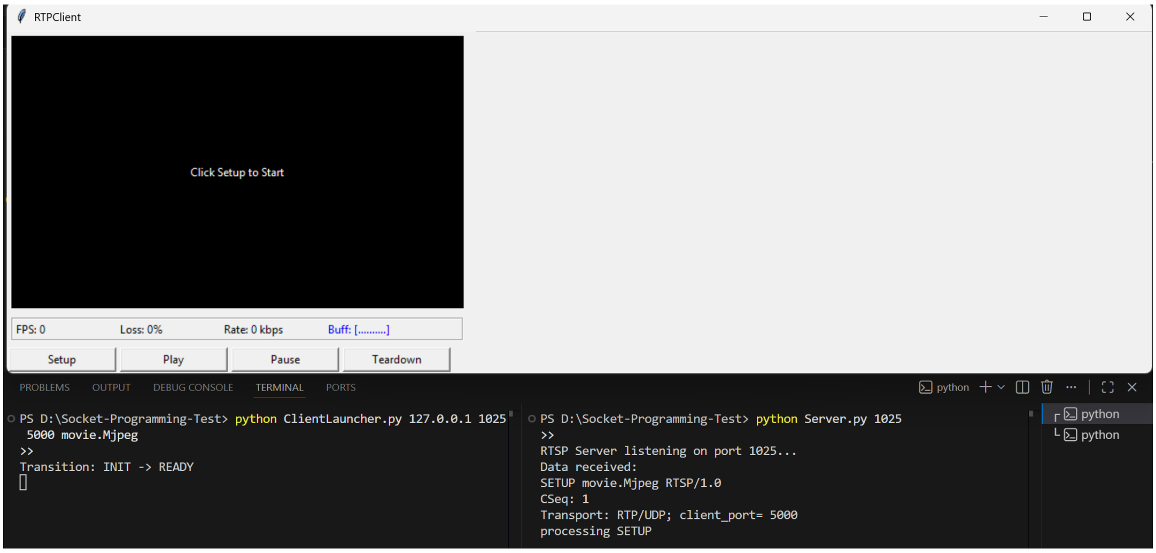

Figure 2: Setup Phase: Log showing SETUP request and 200 OK response

3.3 State 3: Playback and Buffering

Action: User clicks the Play button.

1. Client logs: Data sent: PLAY ... and `Transition: READY -> PLAYING`.
2. Server logs: processing PLAY.
3. Buffering: The cell Buff[....] transform into Buff[==..].
4. Streaming: Once the buffer is ready, the video plays. The terminal prints continuous frame updates.

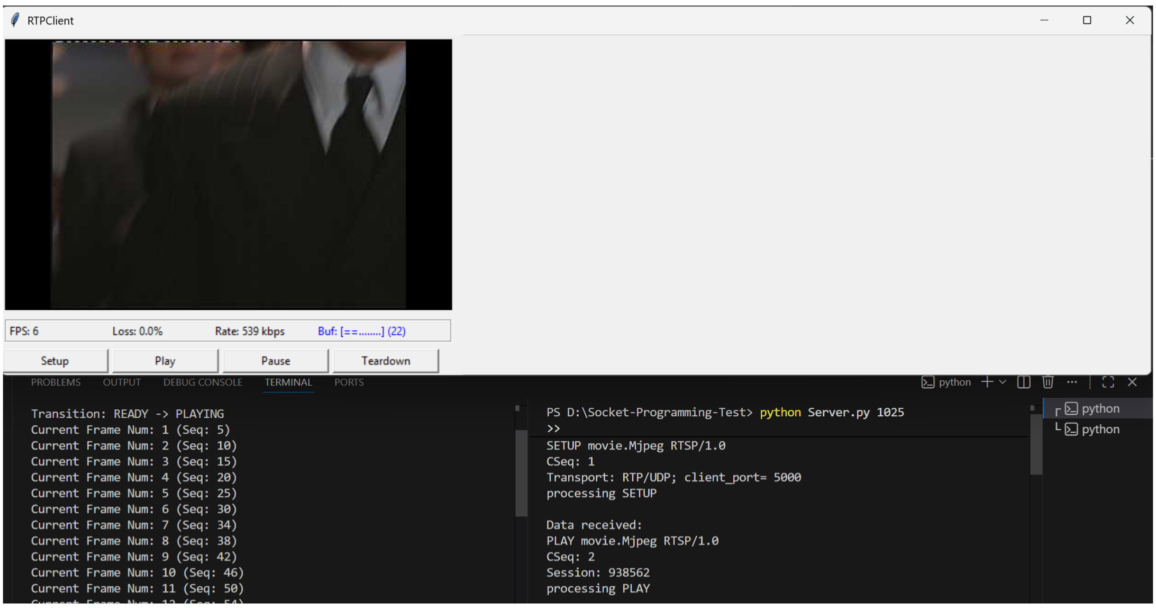

Figure 3: Active Streaming: Video playing with Buffer Bar full and FPS stats

3.4 State 4: Pause and Resume

Action: User clicks Pause, waits, then clicks **Play**.

- Pause: Client sends PAUSE, Server logs processing PAUSE. Client logs: `Transition: PLAYING -> READY`.
- Resume: Client sends PLAY. Video resumes smoothly from the buffer without tearing.

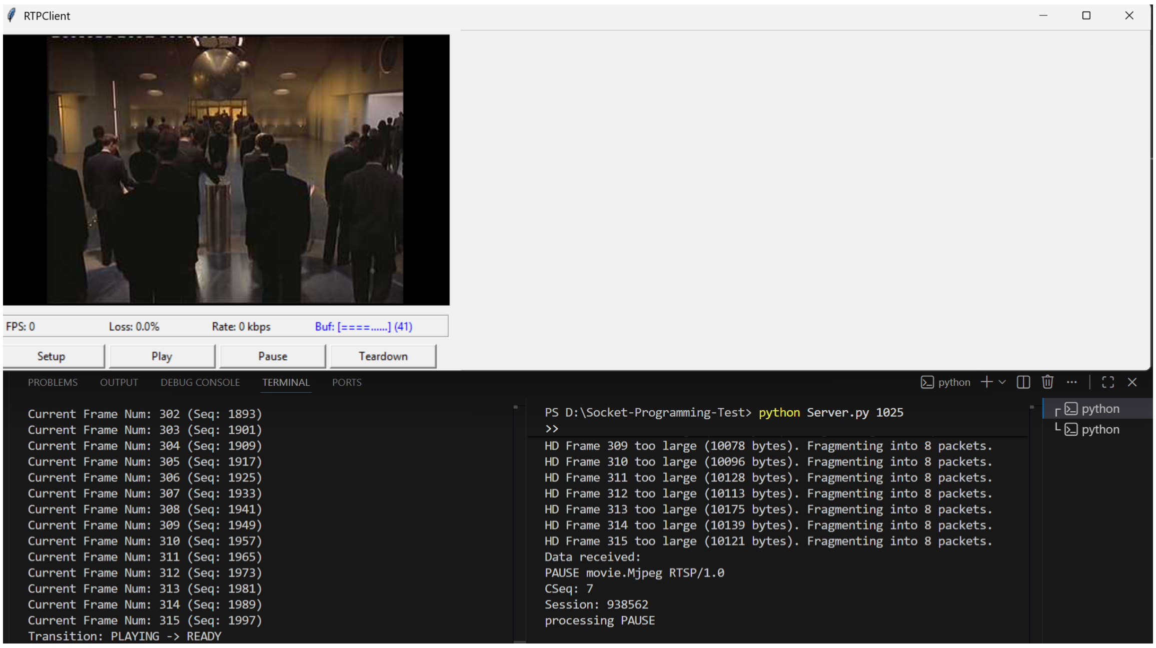

Figure 4: Pause/Resume: Logs verifying state transitions

3.5 State 5: Video Sample 1280x720

- Testing another video with higher resolution
- Fragmenting large frame into many packets to stream

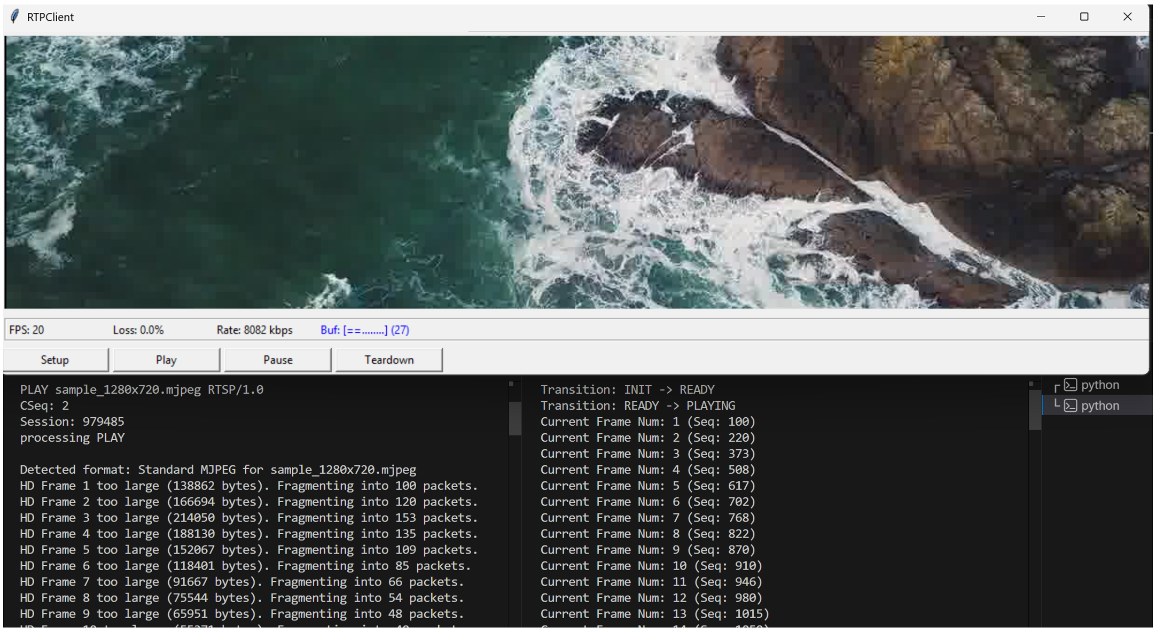

Figure 5: Streaming video with HD (1280x720) resolution

3.6 State 6: Session Teardown

Action: User clicks Teardown.

- Client sends `TEARDOWN`.
- Server logs `processing TEARDOWN`.
- Client logs `Transition: -> INIT` and closes the socket.

Figure 6: Teardown: Log showing session termination

4 Part 4: Contributions and Task Assignment

Student Name	Student ID	Assigned Tasks
Trần Anh Khoa	24127424	Basic video streaming HD video streaming
Lâm Quốc Bảo	20127115	Client-Server Catching GUI & Real-time Statistics Report

Table 1: Group Contributions

5 Conclusion

We have successfully implemented a robust video streaming application that meets all basic and advanced requirements.

- We implemented the complete RTSP State Machine to manage session lifecycles.
- We solved the MTU limitation for HD videos using Application-Layer Fragmentation.
- We ensured smooth playback under network jitter using Client-Side Caching and verified it through real-time GUI statistics.

The project demonstrates a deep understanding of Socket Programming, Bit-level Protocol Design, and Multithreading synchronization.