EE 4020: Introduction to Computer Networks Term Project_Topic1
member:陳彥甫 B11901169 高亮家 B11901130 蔡承岳 B11502067

functionalities:basic with additional features 1, 2, 3
workload: 陳彥甫 33.3%, 高亮家 33.3%, 蔡承岳 33.3%

streaming_client.py

```python
import socket
import threading
import tempfile
import cv2
import os
import time
```

Loads networking, threading, temp-file, OpenCV, filesystem and timing libraries.

```python
SERVER_IP = input("Enter server IP: ").strip()
VIDEO_PORT = int(input("Enter server VIDEO port: ").strip())
CTRL_PORT = int(input("Enter server CTRL port: ").strip())
TEMP_VIDEO = tempfile.NamedTemporaryFile(delete=False, suffix=".mp4").name

stop_flag = threading.Event()
```

Ask user for server IP and two ports.Create a temp-filename for the incoming .mp4. Prepare a thread-safe “stop” event.

```python
def send_control(ctrl_sock):
    while not stop_flag.is_set():
        try:
            cmd = input("Enter command (p=play, t=pause, q=stop): ").strip()
            if stop_flag.is_set():
                break
            if cmd in ['p', 't', 'q']:
                ctrl_sock.sendall(cmd.encode())
                print(f"[CLIENT] Sent command: {cmd}")
                if cmd == 'q':
                    stop_flag.set()
                    break
        except Exception:
            break
```

Reads play/pause/quit from stdin, sends it over the control socket, and triggers shutdown on ‘q’.

```python
def recv_video(video_sock):
    total_bytes = 0
    chunk_count = 0
    first_byte_time = None
    last_byte_time = None
    connect_time = time.time()  # 連線成功的時間
    startup_delay = None  # 啟動延遲時間
```

```python
with open(TEMP_VIDEO, "wb") as f:
    while not stop_flag.is_set():
        try:
            data = video_sock.recv(4096)
            if data:
                if first_byte_time is None:
                    first_byte_time = time.time()
                    startup_delay = first_byte_time - connect_time
                    print(f"[CLIENT] Startup Delay: {startup_delay:.3f} seconds")
                f.write(data)
                total_bytes += len(data)
                chunk_count += 1
                print(f"[CLIENT] Received chunk {chunk_count}: {len(data)} bytes (Total: {total_bytes} bytes)")
            else:
                last_byte_time = time.time()
                break
        except Exception as e:
            print(f"[CLIENT] Video receive error: {e}")
            break
```

```python
print("[CLIENT] Video reception finished.")
if first_byte_time and last_byte_time:
    total_transfer_time = last_byte_time - first_byte_time
    avg_throughput = total_bytes / total_transfer_time if total_transfer_time > 0 else 0
    print(f"[CLIENT] Startup Delay: {startup_delay:.3f} seconds")
    print(f"[CLIENT] Total Transfer Time: {total_transfer_time:.3f} seconds")
    print(f"[CLIENT] Average Throughput: {avg_throughput/1024:.2f} KB/s")
stop_flag.set()
```

Streams raw bytes into the temp file.
Logs first-byte delay, per-chunk sizes, and overall transfer time/throughput.

```python
def main():
    video_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    ctrl_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    video_sock.connect((SERVER_IP, VIDEO_PORT))
    ctrl_sock.connect((SERVER_IP, CTRL_PORT))
    print("[CLIENT] Connected to server (video & control)")

    t1 = threading.Thread(target=recv_video, args=(video_sock,))
    t2 = threading.Thread(target=send_control, args=(ctrl_sock,))
    t1.start()
    t2.start()
    t1.join()
    stop_flag.set()
    video_sock.close()
    ctrl_sock.close()
    print("[CLIENT] Connections closed.")
```

Opens two TCP sockets (video + control). Spins off receiver & sender threads. Waits for streaming to finish, then tears down.

```python
    cap = cv2.VideoCapture(TEMP_VIDEO)
    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            break
        cv2.imshow('Received Video', frame)
        if cv2.waitKey(30) & 0xFF == ord('q'):
            break
    cap.release()
    cv2.destroyAllWindows()
    os.remove(TEMP_VIDEO)

if __name__ == "__main__":
    main()
```

Once download completes, opens the temp .mp4 with OpenCV and plays it back. On exit, cleans up the file and window.

streaming_server.py

```python
import socket
import threading
import os
import argparse
import time

VIDEO_FILE = "sample.mp4"
CHUNK_SIZE = 4096

client_states = {}
lock = threading.Lock()
```

Imports modules, sets video file and chunk size, and prepares shared state and lock for multi-threaded streaming.

```python
def handle_control(conn, addr):
    global client_states
    try:
        while True:
            cmd = conn.recv(1024).decode().strip()
            if not cmd:
                break
            with lock:
                if cmd == 'p':
                    client_states[addr]['state'] = 'playing'
                    print(f"[SERVER] {addr} PLAY received. State=PLAYING")
                elif cmd == 't':
                    client_states[addr]['state'] = 'paused'
                    print(f"[SERVER] {addr} PAUSE received. State=PAUSED")
                elif cmd == 'q':
                    client_states[addr]['state'] = 'stopped'
                    print(f"[SERVER] {addr} STOP received. State=STOPPED")
                    break
    except Exception as e:
        print(f"[SERVER] Control connection error: {e}")
    finally:
        conn.close()
        print(f"[SERVER] Control connection with {addr} closed.")
```

This function receives control commands (p, t, q) from the client over the control connection.

It updates the client's streaming state (playing, paused, stopped) in a thread-safe way using a lock, prints the command and new state, and closes the control connection when finished or if an error occurs.

This allows the server to respond to client playback controls in real time.

```
def handle_video(conn, addr, delay):
    global client_states
    try:
        with open(VIDEO_FILE, "rb") as f:
            while True:
                with lock:
                    state = client_states[addr]['state']
                if state == 'playing':
                    chunk = f.read(CHUNK_SIZE)
                    if not chunk:
                        break
                    try:
                        conn.sendall(chunk)
                        print(f"[SERVER] Sent chunk to {addr}: {len(chunk)} bytes")
                    except:
                        break
                    if delay > 0:
                        time.sleep(delay)
                elif state == 'paused':
                    time.sleep(0.1)
                elif state == 'stopped':
                    break
        print(f"[SERVER] Video streaming to {addr} finished.")
    except Exception as e:
        print(f"[SERVER] Video connection error: {e}")
```

```
    finally:
        conn.close()
        with lock:
            client_states.pop(addr, None)
        print(f"[SERVER] Video connection with {addr} closed.")
```

This function streams the video file to the client in chunks.

It checks the client's state (playing, paused, stopped) and only sends data when the state is playing.

If paused, it waits; if stopped or the file ends, it breaks the loop.

It also adds a delay between chunks if specified, and cleans up the connection and client state when finished.

```python
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--delay', type=float, default=0.0, help='Delay (in seconds) between sending chunks')
    args = parser.parse_args()
    delay = args.delay

    # 1. 檢查影片檔案
    if not os.path.exists(VIDEO_FILE):
        print(f"[SERVER] Error: {VIDEO_FILE} not found.")
        return
    print(f"[SERVER] Found video file: {VIDEO_FILE}")

    # 2. 監聽兩個 port
    video_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    ctrl_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    video_sock.bind(('', 0))
    ctrl_sock.bind(('', 0))
    video_sock.listen(5)
    ctrl_sock.listen(5)
    video_port = video_sock.getsockname()[1]
    ctrl_port = ctrl_sock.getsockname()[1]
    print(f"[SERVER] Listening on VIDEO port {video_port}, CTRL port {ctrl_port} (delay={delay}s)")
```

This part parses the command-line delay argument, checks if the video file exists, and sets up two TCP sockets to listen for incoming video and control connections on separate ports.

It prints the port numbers and delay value for the user to connect the client.

```python
    while True:
        print("[SERVER] Waiting for video connection...")
        v_conn, v_addr = video_sock.accept()
        print(f"[SERVER] Client {v_addr} connected (video)")
        print("[SERVER] Waiting for control connection from same client...")
        c_conn, c_addr = ctrl_sock.accept()
        print(f"[SERVER] Client {c_addr} connected (control)")

        # 初始化狀態為 paused
        with lock:
            client_states[v_addr] = {'state': 'paused'}

        # 啟動控制與資料 thread
        threading.Thread(target=handle_control, args=(c_conn, v_addr), daemon=True).start()
        threading.Thread(target=handle_video, args=(v_conn, v_addr, delay), daemon=True).start()
```

This loop waits for a new client to connect on both the video and control sockets.

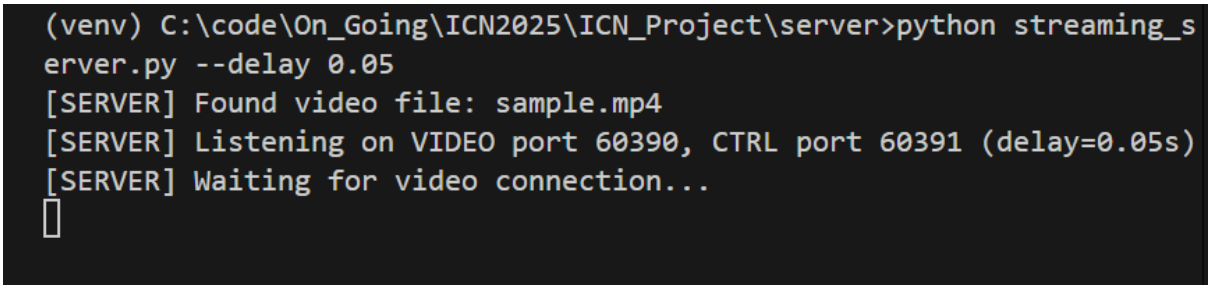
It initializes the client's state as paused, then starts two threads: one to handle control commands and one to stream video data to the client.

This allows each client to have independent playback control and streaming.

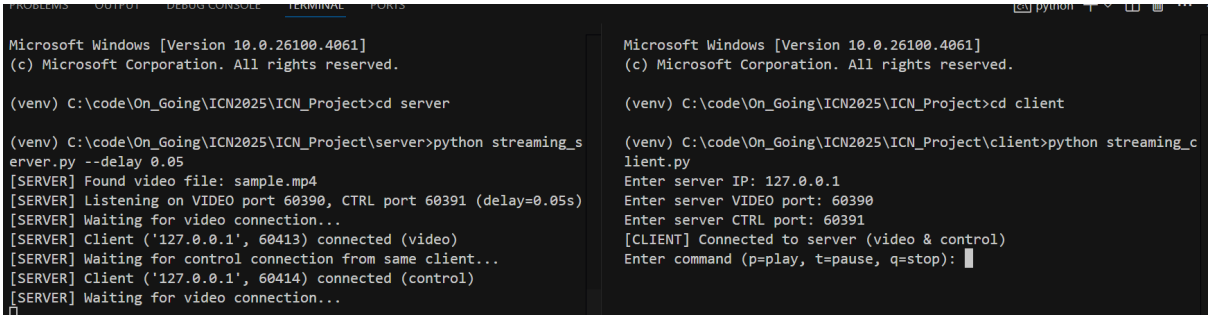
Basic Functionality

Demonstration:

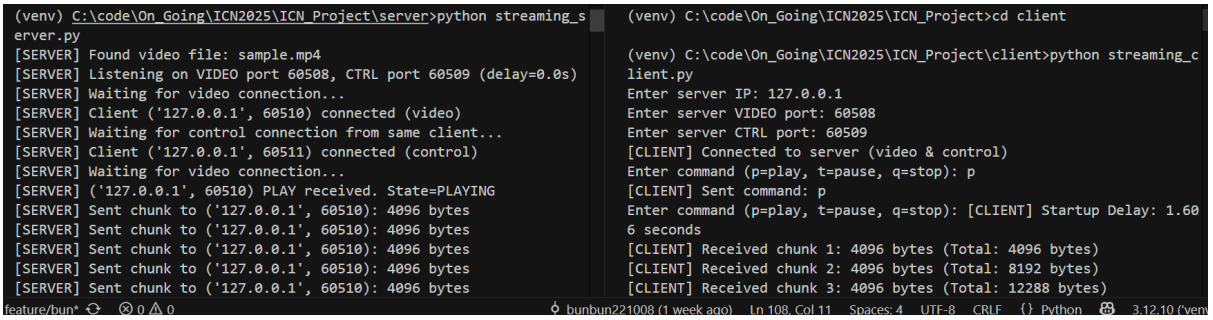
(1) Show the streaming_server.py running and confirming it has access to ‘sample.mp4‘.



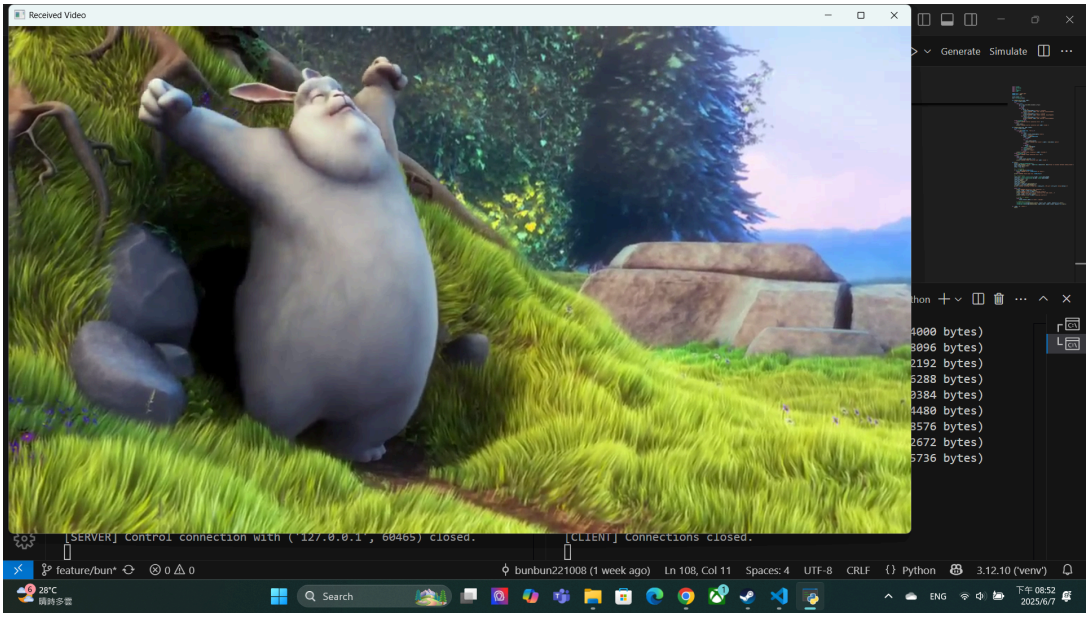
(2) Show the streaming_client.py running and successfully connecting to the server.



(3) Show logs from both client and server indicating the ‘Simple Streaming Protocol Trigger‘ occurred (e.g., client sent ‘START‘, server received ‘START‘) and the video streaming starts.

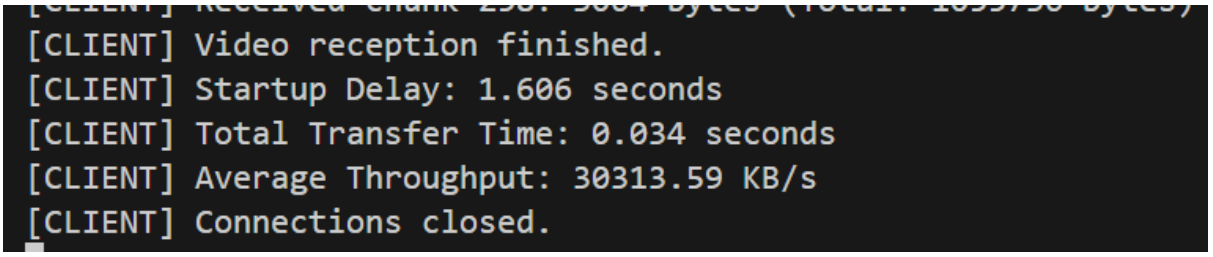


(4) Show that the client is displaying frames from the ‘sample.mp4‘ video that was transferred over the network.



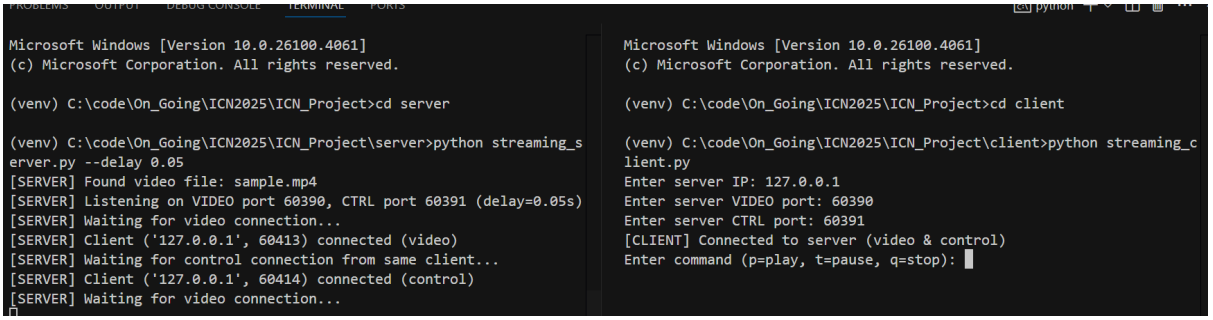
Basic Performance Metrics Logging

Record three of the following metrics on the client-side:

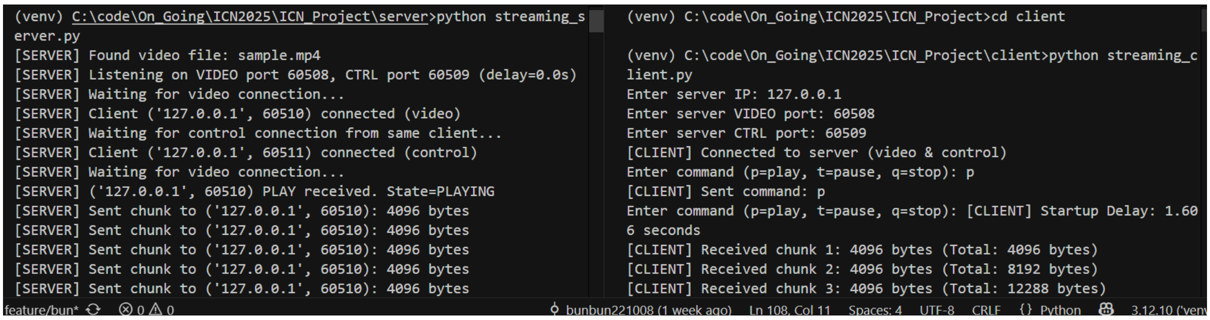


Demonstration:

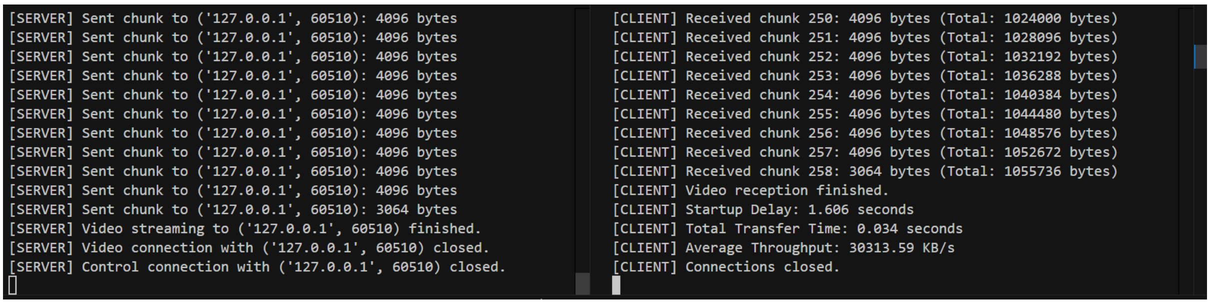
(a) Run streaming_server.py first and then run streaming_client.py.



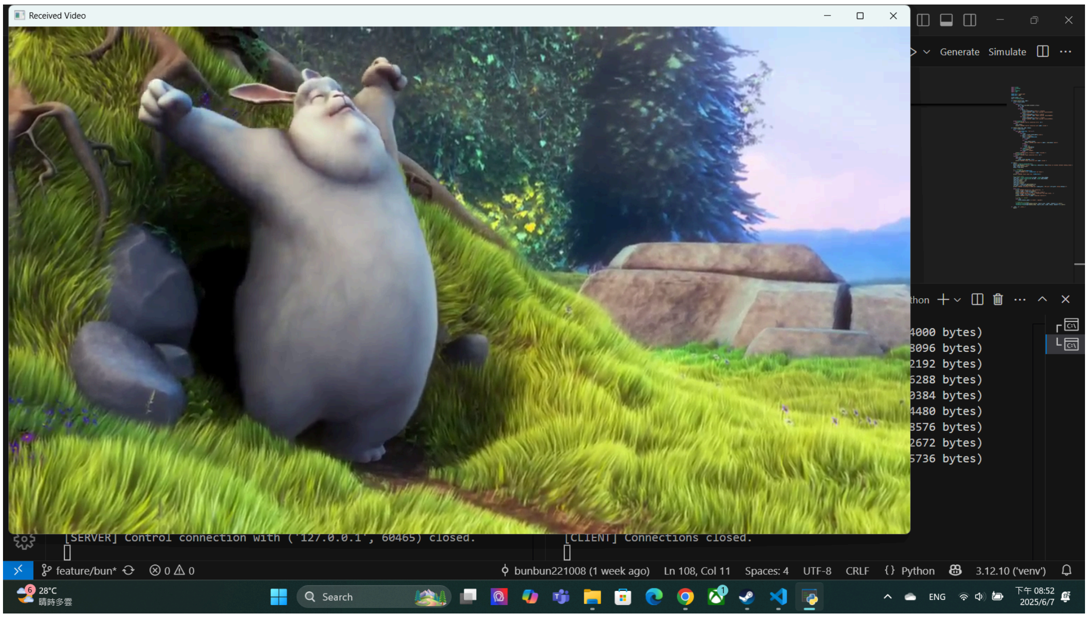
(b) Show logs from both client and server indicating the ‘Simple Streaming Protocol Trigger‘ occurred (e.g., client sent ‘START‘, server received ‘START‘).



(c) Show logs from the server indicating it is sending data chunks and logs from the client indicating it is receiving data chunks.



(d) Show a screenshot of the OpenCV window launched by streaming_client.py to display frames from the ‘sample.mp4‘ video.



Simple Rate Control Simulation:

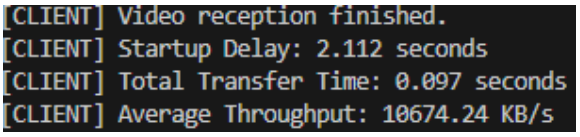
Demonstration Steps:

(a) Run the server with no added delay. Run the client and observe/record the playback smoothness and the logged the average throughput .

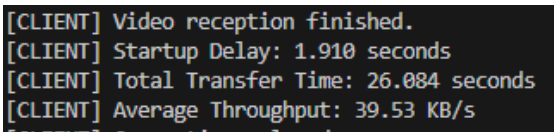
(b) Stop the server and restart it with a noticeable delay (e.g., ‘–delay 0.1‘).

(c) Run the client again. Show that the playback is noticeably slower compared to the run with no delay. Also, show the logged average Throughput is significantly lower.

(d) Compare the results of both runs in your report.



a. results with no delay

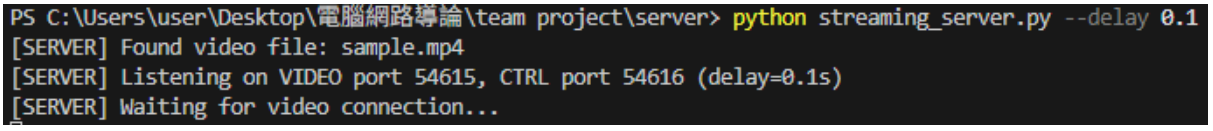


b. results with 0.1s delay

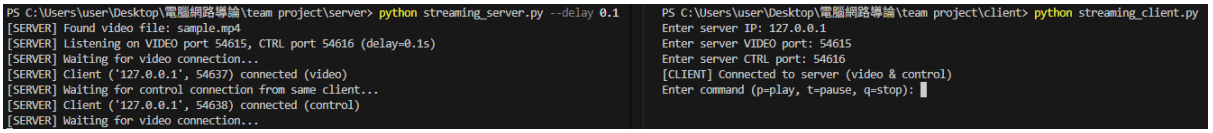
Basic Player Control Protocol:

Demonstration Steps:

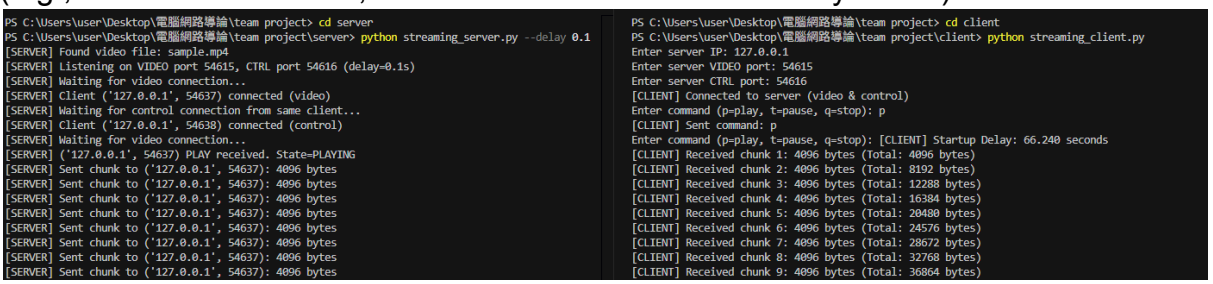
(a) Start Server with Delay: Run the ‘streaming_server.py‘ using the ‘–delay‘ argument with a large delay (e.g., ‘python streaming_server.py –delay 0.1‘) to allow time for interaction. Show logs indicating it’s listening on both Video Connection and Control Connection.



(b) Connect Client: Run ‘streaming_client.py‘. Show logs confirming both Video Connection and Control Connection are established.



(c) Start Playback (‘p‘ command): Press ’p’ on the client terminal. Show client log confirming ‘PLAY‘ command was sent. Show server log confirming ‘PLAY‘ received and state changed to ‘PLAYING‘. Show client log indicating it starts receiving data (e.g., "Received chunk 1", "Received chunk 2"... or "Total bytes: X").



(d) Test Pause Mid-Stream (‘t‘ command): While data chunks are arriving (client counter is increasing), press ’t’ on the client terminal. - Show client log confirming ‘t‘ command sent. - Show server log confirming ‘t‘ received‘. - Show that the client’s chunk/byte counter stops increasing, indicating data flow has paused.

(e) Test Resume Mid-Stream (‘p‘ command): While paused, press ’p’ again on the client terminal. - Show client log confirming ‘p‘ command sent. - Show server log confirming ‘p‘ received. - Show that the client’s chunk/byte counter resumes increasing, indicating data flow has resumed.

```
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] ('127.0.0.1', 54637) PAUSE received. State=PAUSED
[SERVER] ('127.0.0.1', 54637) PLAY received. State=PLAYING
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
```

```
[CLIENT] Sent command: t
Enter command (p=play, t=pause, q=stop): p
[CLIENT] Sent command: p
Enter command (p=play, t=pause, q=stop): [CLIENT] Received chunk 13: 4096 bytes (Total: 53248 byte
s)
[CLIENT] Received chunk 14: 4096 bytes (Total: 57344 bytes)
[CLIENT] Received chunk 15: 4096 bytes (Total: 61440 bytes)
[CLIENT] Received chunk 16: 4096 bytes (Total: 65536 bytes)
[CLIENT] Received chunk 17: 4096 bytes (Total: 69632 bytes)
[CLIENT] Received chunk 18: 4096 bytes (Total: 73728 bytes)
[CLIENT] Received chunk 19: 4096 bytes (Total: 77824 bytes)
[CLIENT] Received chunk 20: 4096 bytes (Total: 81920 bytes)
[CLIENT] Received chunk 21: 4096 bytes (Total: 86016 bytes)
[CLIENT] Received chunk 22: 4096 bytes (Total: 90112 bytes)
[CLIENT] Received chunk 23: 4096 bytes (Total: 94208 bytes)
[CLIENT] Received chunk 24: 4096 bytes (Total: 98304 bytes)
[CLIENT] Received chunk 25: 4096 bytes (Total: 102400 bytes)
```

(f) Test Stop (‘q‘ command): Send ’q’ from the client. Show client log confirming ‘q‘ command sent. Show server log confirming ‘q‘ received. Show logs from both client and server indicating both connections closed and the client/server threads handling this session exited cleanly (chunk counter stops definitively).

```
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] Sent chunk to ('127.0.0.1', 54637): 4096 bytes
[SERVER] ('127.0.0.1', 54637) STOP received. State=STOPPED
[SERVER] Control connection with ('127.0.0.1', 54637) closed.
[SERVER] Video streaming to ('127.0.0.1', 54637) finished.
[SERVER] Video connection with ('127.0.0.1', 54637) closed.
```

```
[CLIENT] Received chunk 65: 4096 bytes (Total: 266240 bytes)
[CLIENT] Received chunk 66: 4096 bytes (Total: 270336 bytes)

[CLIENT] Sent command: q
[CLIENT] Video reception finished.
[CLIENT] Startup Delay: 66.240 seconds
[CLIENT] Total Transfer Time: 420.565 seconds
[CLIENT] Average Throughput: 0.63 KB/s
[CLIENT] Connections closed.
[mov,mp4,m4a,3gp,3g2,mj2 @ 000001901386c780] moov atom not found
```