REDES

PRÁCTICA I

Programación de aplicaciones de red

Daranuta Zop Cristian

55261830Y

GPraLab1

07 / 05 / 2025

INDEX

Introducción

Este proyecto consiste en el desarrollo de un sistema de transmisión de vídeo en tiempo real basado en los protocolos RTSP y RTP, implementado en Python. Para ello, se han creado principalmente dos clases: una para el servidor, que gestiona sesiones y envía vídeo mediante datagramas UDP, y otra para el cliente, que cuenta con una interfaz gráfica capaz de recibir, decodificar y mostrar los frames del vídeo en tiempo real.

El objetivo principal es lograr que el servidor sea capaz de responder correctamente a las peticiones del cliente y enviar el flujo de vídeo por UDP, permitiendo además la conexión simultánea de varios clientes sin interferencias entre ellos. Por su parte, el cliente ofrece una interfaz gráfica intuitiva que permite al usuario interactuar mediante botones, controlar la reproducción y visualizar el vídeo en tiempo real.

PARTE I. cli.py

Dependencias

- **click** - Biblioteca de procesamiento de parámetros de línea de órdenes. La utilizo en el código que te he dado de ejemplo.
- **loguru** - Biblioteca que reemplaza el logging de Python. La he utilizado en el programa que os he pasado.
- **sys -** Biblioteca para acceder a variables y funciones que controlan el comportamiento del intérprete de Python.
- **server.Server** - Importación de la clase Server
- **client.Client** - Importación de la clase Client

Comando server

Comando:

```
xarxes2025 server [OPTIONS]
```

Descripción:

- Inicia un servidor RTSP que transmite vídeo por RTP/UDP.

Opciones:

- `--port`, `-p`: Puerto TCP donde escuchar conexiones RTSP (por defecto: 4321).
- `--host`, `-h`: IP donde levantar el servidor (por defecto: 127.0.0.1).
- `--max-frames`: Número máximo de frames a transmitir. `-1` para sin límite.
- `--frame-rate`: Número de frames por segundo (por defecto: 25).
- `--loss-rate`: Tasa de pérdida simulada en el envío por UDP (0–100).
- `--error`: Simula condiciones de error específicas (por ejemplo, caídas, retardos).

Ejemplo de uso:

```
xarxes2025 server --port 4321 --frame-rate 30 --loss-rate 10
```

Comando client

Comando:

```
xarxes2025 client [VIDEOFILE] [OPTIONS]
```

Descripción:

- Inicia un cliente RTSP que se conecta al servidor y recibe vídeo por RTP.

Argumentos:

- `videofile`: Ruta del archivo de vídeo que se desea reproducir (por defecto: `rick.webm`).

Opciones:

- `--port`, `-p`: Puerto TCP al que conectarse en el servidor RTSP (por defecto: 4321).
- `--host`, `-h`: Dirección IP del servidor RTSP (por defecto: 127.0.0.1).
- `--udp-port`, `-u`: Puerto UDP en el cliente para recibir el vídeo por RTP (por defecto: 25000).

Ejemplo de uso:

```
xarxes2025 client rick.webm --udp-port 26000
```

PARTE II. UDPDatagram Class

Esta clase, implementa la construcción y el análisis de los datagramas RTP (Real-time Transport Protocol), los cuales son utilizados en la aplicaciones de transmisión de medios en tiempo real sobre redes UDP.

El objetivo es permitir la creación, envío y recepción de paquetes en el estándar RTP, encapsulando *header* y *payload*

Estructura del paquete RTP

Los datagramas constan de dos partes:

- **Cabecera (*Header*):** Tamaño de 12 bytes, contiene metadatos:
- ***Payload*:** Contiene los datos de la imagen.

Constructor

Constructor de la clase, que inicia los parámetros de los Datagramas RTP

Parámetros:

- `seqnum` - Número de secuencia del paquete, usado para ordenar los paquetes recibidos.
- `payload` - Los datos reales a enviar
- `ssrc` - Identificador único de la fuente emisora. Por defecto se usa 0.

Métodos

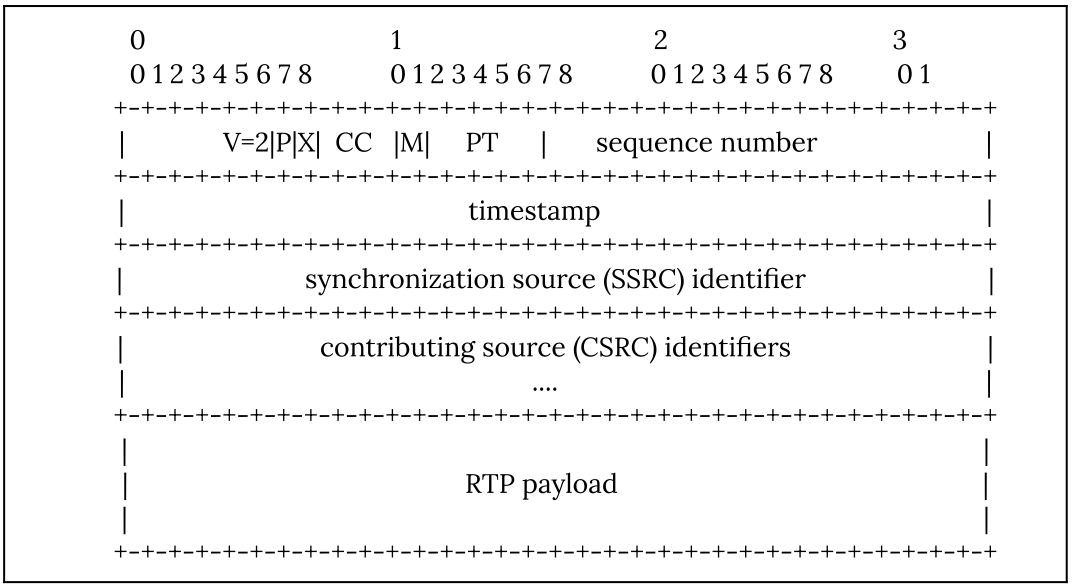
encode

Construir la cabecera RTP e insertar el payload. Este método es llamado automáticamente por el constructor.

Cabecera codificada:

- `Versión`: 2 bits valor estándar RTP
- `Padding`: 0 bits
- `Extension`: 0 bits
- `CC`: 0 bits
- `Marker`: 0 bits
- `Payload Type (PT)`: 26 bits, usado para MJPEG
- `Sequence Number`: Número secuencial de 16 bits para reconstrucción en el receptor.
- `Timestamp`: Marca de tiempo basada en `time()` de Python
- `SSRC`: Identificador de 32 bits para distinguir múltiples flujos.

El formato de este paquete es el siguiente:



decode

Método para extraer la cabecera y el *payload* a partir de un flujo de bytes recibido a partir de un socket UDP. Es invocado por el cliente, para poder decodificar los datagramas entrantes

get_version

Método para devolver la versión del protocolo desde los bits más significativos del primer byte de la cabecera.

get_seqnum

Método para devolver el número de secuencia a partir de los bytes 2 y 3 de la cabecera. Es útil para ordenar los paquetes en el cliente receptor.

timestamp

Método que devuelve el *timestamp* del paquete (bytes 4 a 7). Indica cuándo se generó el paquete.

get_payload

Método que devuelve el contenido binario del *payload*, es decir, los datos de imagen que se transmiten.

get_datagram

Método que devuelve el datagrama completo (*header* + *payload*) en un solo bloque de bytes. Asi lo podemos enviar por el *socket* UDP.

PARTE III. VideoProcessor Class

Dependencias

- **cv2**: Biblioteca OpenCV para procesamiento de imágenes y vídeo.
- **loguru.logger**: Utilizado para registrar eventos como la apertura de archivos y errores.

Atributos de Clase

```
ready = False
```

Atributo que se usa para indicar si el video esta listo para ser usado una vez abierto el archivo correctamente.

Constructor

Constructor de la clase, que inicia el objeto para procesar el video

Parámetros:

- `filename` — Nombre o ruta del archivo de vídeo a abrir.

Funcionamiento:

Se intenta abrir el archivo mediante la librería ***cv2***

- En caso de que sea exitoso, el atributo `ready` se pone a `True`
- Si no se pudo abrir, salta una excepcion (`IOError`)

Métodos

next_frame

Metodo usado para leer y retornar el siguiente frame del video, en formato de imagen *JPEG*, codificada y lista para ser enviada mediante *RTP/UDP*.

Funcionamiento:

1. Lee el siguiente fotograma.
2. Si no hay más fotogramas (fin del vídeo), retorna `None`.
3. Redimensiona el fotograma a `500x380` píxeles. (Este tamaño se debe a que es ideal para enviarlo mediante UDP sin fraccionamiento)
4. Codifica el fotograma en formato JPEG.
5. Retorna los bytes codificados.

Return:

- `bytes` — Fotograma JPEG codificado en bytes.
- `None` — Si se alcanza el final del vídeo.

Errores:

- Lanza `IOError` si ocurre un error durante la codificación del fotograma.

get_frame_number

Método que retorna el número del fotograma actual procesado.

PARTE III. Server Class

En esta clase Server, se implementa un servidor RTSP, para streaming de video, mandado sobre TCP y UDP. Este mismo, acepta conexiones RTSP entrantes, y también gestiona sesiones de clientes, y a su vez, retransmite videos en formato RTP encapsulados en Datagramas UDP.

Dependencias

- **socket:** Módulo estándar de Python para la comunicación en red usando sockets (TCP/UDP).
- **threading:** Biblioteca estándar de Python para ejecutar tareas en paralelo usando hilos.
- **time:** Módulo estándar de Python para manejar operaciones relacionadas con el tiempo. Usado para controlar la velocidad de envío de los paquetes RTP simulando la reproducción del vídeo
- **re**: Biblioteca para parseo de respuestas RTSP.
- **random:** Librería para generar numeros random usada para generar sesiones
- **loguru**: Librería que reemplaza la librería **loggin** de Python.
- **UDPDatagram**: Construcción de paquetes UDP/RTP
- **VideoProcessor**: Obtención de frames del vídeo

Estados (Enum)

Máquina de estados que controla la evolución de la sesión de vídeo. Cada cliente gestiona su propio estado local.

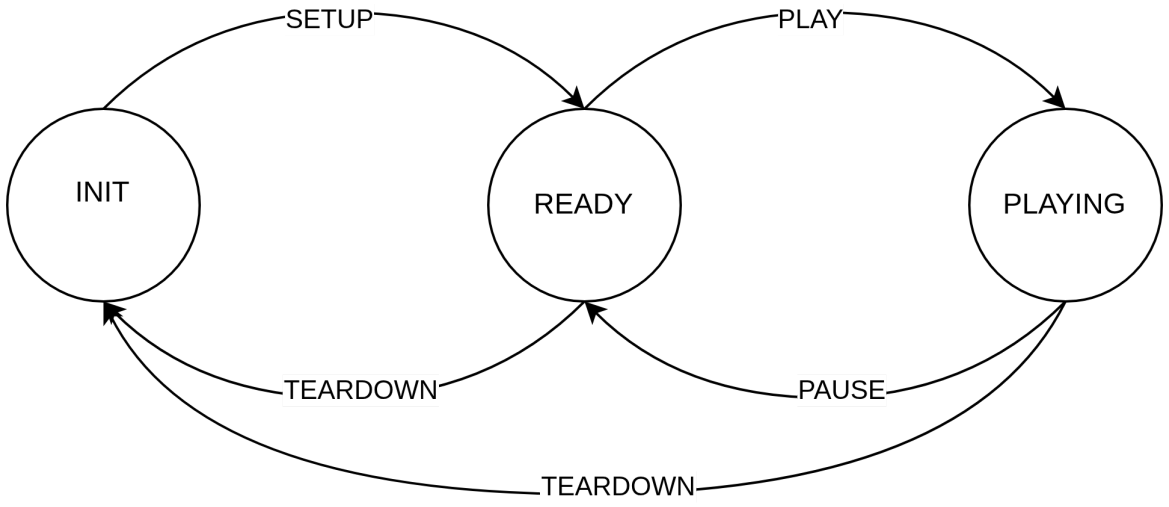


Figura 1: Máquina de estados implementada para el protocolo RTSP Servidor

Constructor

Parámetros:

- **port**: Puerto TCP donde el servidor escucha conexiones RTSP.
- **host**: Dirección IP del servidor.
- **max_frames**: Número máximo de frames que el servidor enviará por sesión.
- **frame_rate**: Velocidad de transmisión en FPS.
- **loss_rate**: Tasa de pérdida de paquetes simulada (0–100%).
- **error**: Código de error para simular fallos RTSP/RTP.

Métodos

run_server (Método Principal)

Método que inicia un socket TCP en el servidor, y una vez hecho, entra en un bucle infinito donde va aceptando clientes. Con tal de poder gestionar varias conexiones a la vez, crea un hilo (*Thread*), y una vez hecho, se le deriva al método **handle_client**.

handle_client

Método implementado para gestionar las conexiones RTSP de los clientes con el servidor. Dado a que se hizo *Threading*, cada cliente se gestiona por separado.

Estados locales manejados:

- `INIT`: Esperando `SETUP`
- `READY`: Preparado para `PLAY`
- `PLAYING`: Transmitiendo vídeo por UDP

Comandos RTSP procesados:

- `SETUP`: Inicializa una sesión y carga el vídeo.
- `PLAY`: Comienza la transmisión de frames por UDP.
- `PAUSE`: Detiene temporalmente la transmisión.
- `TEARDOWN`: Finaliza la sesión actual y libera recursos.
- `CLOSE`: Cierra conexión TCP y termina el hilo.
- Otros comandos generan error 501 (Not Implemented).

Gestión adicional:

- Los clientes tienen un puerto UDP único y exclusivo.

- Soporte de simulación de errores en respuestas RTSP

send_udp_frames

Método implementado para enviar los frames del video procesando por un puerto UDP al cliente siempre que `is_active()` retorne True. La transmisión respeta el tema de `frame_rate` y `max_frames.`

Lógica adicional:

- Fragmenta cada frame usando `UDPDatagram`.
- Puede simular pérdida de paquetes basada en `loss_rate`.
- Finaliza al alcanzar el número máximo de frames o si no hay más datos.

send_rtsp_response

Método para enviar respuestas RTSP al cliente mediante el código de estado, CSeq y el número de sesión.

Mensajes soportados:

- 200 OK
- 400 Bad Request
- 404 File Not Found
- 500 Internal Server Error
- 501 Not Implemented

Soporte de errores simulados:

- error == 100: Session ID incorrecto
- error == 101: CSeq erróneo (valor -1)
- error == 102: CSeq como texto ("ERROR")

PARTE IV. Client Class

En esta clase se implementa un cliente RTSP/RTP, que mediante una interfaz gráfica, permite conectarse a un servidor mediante TCP(RTSP), controlar la reproducción de video gracias a comandos RTSP y visualizar el video mediante frames recibidos por un puerto UDP, los cuales están encapsulados en datagramas.

Dependencias

- **socket:** Módulo estándar de Python para la comunicación en red usando sockets (TCP/UDP).
- **threading**: Biblioteca estándar de Python para ejecutar tareas en paralelo usando hilos.
- **PIL.Image, ImageTk**: Módulo para mostrar los frames como imágenes.
- **loguru**: Libreria para registrar toda la actividad.
- **tkinter**: Biblioteca para generar la interfaz de usuario.
- **re**: Biblioteca para parseo de respuestas RTSP.
- **UDPDatagram**: Decodifica el contenido de los datagramas RTP

Estados (Enum)

De igual modo que en el servidor, para el cliente también se le implementa una máquina de estados, con tal de hacer más fácil la gestión de estados y evitar errores o tratamientos complejos.

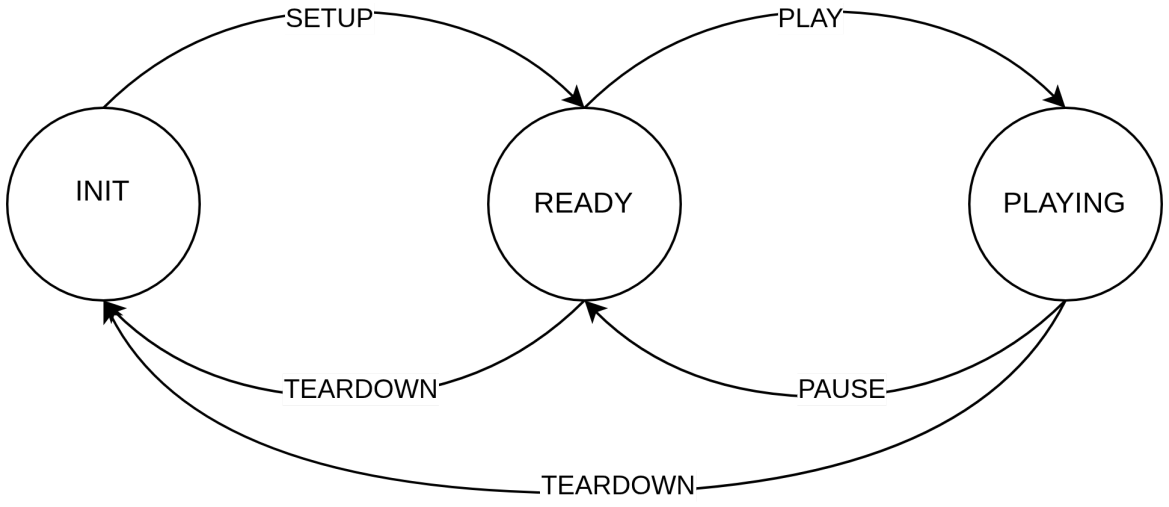


Figura 2: Máquina de estados implementada para el protocolo RTSP Cliente

Constructor

Parámetros:

- **server_port**: Puerto TCP donde el cliente mandara las peticiones RTSP al servidor.
- **server_host**: Dirección IP del servidor.
- **udp_port**: Puerto UDP donde el cliente recibirá los frames del video que el servidor le reproducirá.
- **filename**: Nombre del archivo de video que se quiere reproducir por el cliente.

Una vez iniciado, el cliente:

- Crea la interfaz gráfica.
- Intenta establecer la conexión TCP con el servidor.

Interfaz gráfica

create_ui

- Crea la ventana principal (`Tk()`).
- Añade botones: `Setup, Play, Pause, Teardown`.
- Añade un *Label* (`self.movie`) para mostrar los frames recibidos.
- Añade otro *Label* (`self.text`) para mostrar mensajes o respuestas del servidor.

_create_button

- Ayuda a crear y ubicar un botón con una función específica.

ui_close_window

- Pregunta al usuario si quiere cerrar.
- Destruye el socket UDP y cierra la ventana si se confirma.

Eventos de los botones

Cada uno de los botones genera una petición RTSP, con los datos correspondientes, la cual se envía al servidor y se pone a la espera de una respuesta.

ui_setup_event

- Estado requerido: INIT
- Envía una petición SETUP
    - Si es positiva, cambia de estado a `READY` y se guarda el número de session

ui_play_event

- Estado requerido: READY
- Envía una petición PLAY
    - Si es positiva, cambia de estado a `PLAYING` y se mostrará por pantalla los frames

ui_pause_event

- Estado requerido: PLAYING
- Envía una petición PAUSE
    - Si es positiva, cambia de estado a `READY` y se dejará de reproducir los frames

ui_teardown_event

- Estado requerido: READY o PLAYING
- Envía una petición TEARDOWN
    - Si es positiva, cambia de estado a `INIT` y se borra la sesion

Conexión y comunicación RTSP

run_client

- Método que inicia una conexión TCP con el servidor

send_request

- Método usado para enviar las peticiones RTSP al servidor mediante el puerto TCP del servidor.
- Una vez recibida la respuesta, es mandada al método `handle_response()`.

handle_response

- Método usado para procesar de forma correcta las respuestas del servidor.
    - Se extrae el código de estado
    - Se extrae también el número de sesión generado por el servidor.
    - Cambia el estado del cliente según el tipo de comando.

Recepción de vídeo por UDP (RTP)

start_udp_listener

- Método usado para escuchar la transmisión de paquetes RTP enviados por el servidor.
    - Crea un socket UDP en el puerto `self.udp_port`.
    - Lanza un `thread` que escucha constantemente.

- Cada datagrama recibido se decodifica usando `UDPDatagram`.
- El *`payload`* se interpreta como imagen y se muestra.

teardown_udp_socket

- Método usado para cerrar el *socket* UDP y desactivar también el *listener*.

Conclusión

Aunque este proyecto es mejor abordarlo con calma y tiempo, sobre todo para investigar y entender cómo funcionan las dependencias, las clases y los distintos componentes, representa un reto realmente interesante. Es una excelente oportunidad para aprender el funcionamiento de protocolos como TCP y UDP, así como la implementación de protocolos más específicos como RTSP y RTP, el encapsulamiento de paquetes, y otros aspectos clave del desarrollo de aplicaciones de red.

En mi caso, quedé completamente satisfecho con el resultado obtenido, y considero que este proyecto deja una base sólida para futuras mejoras o ampliaciones. Eso sí, es fundamental no dejarlo para el último momento ni intentar resolverlo en dos días, ya que requiere bastante depuración, prueba y error.

Bibliografía

XARXES (102015-2425) - Campus Virtual UdL. (s.f.). https://cv.udl.cat/portal/site/102015-2425/tool/c4de53bd-8360-4ba9-9930-7a921e971e88?panel=Main

Orlivskyi, S. (2024, 29 de abril). Guía completa de programación de sockets en Python. https://www.datacamp.com/es/tutorial/a-complete-guide-to-socket-programming-in-python

socket (Low-level networking interface. (s.f.). Python documentation. https://docs.python.org/3/library/socket.html

Colaboradores de los proyectos Wikimedia. (2006, 18 de noviembre). Protocolo de transmisión en tiempo real - Wikipedia, la enciclopedia libre. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Protocolo_de_transmisión_en_tiempo_real