Redes Multimedia – Prácticas
Práctica 2: Medidas de rendimiento

Turno y pareja: **3412_13**

Integrantes:

Guillermo Sánchez Esteban

Diego Quintero Muñoz

Fecha de entrega: 12/11/2025

Contenido

1 Introducción

El objetivo de esta práctica es analizar el comportamiento de una transmisión de datos basada en el envío de trenes de paquetes UDP, aplicando una cabecera simplificada de tipo RTP. A partir de estos envíos se pretende medir parámetros característicos de una comunicación en red, como el retardo en un sentido, la variación del retardo (jitter), el ancho de banda y la pérdida de paquetes.

Para ello se han desarrollado dos programas en Python:

- **clienteTren.py**, encargado de generar y enviar los trenes de paquetes, configurando la longitud del tren y el tamaño de los datos;
- **servidorTren.py**, encargado de recibir los paquetes y calcular las métricas anteriores a partir de los tiempos de envío y recepción.

Durante la práctica se realizan pruebas tanto en la **interfaz local (loopback)** como en una **red de área local (LAN)** entre dos equipos, con el fin de observar las diferencias de comportamiento debidas a la presencia o no de las cabeceras de capa de enlace (Ethernet) y a las condiciones reales de transmisión.

Los resultados obtenidos permiten comprender cómo afectan la longitud de los paquetes, el tamaño del tren y las cabeceras de las distintas capas al rendimiento global de la comunicación, así como validar las fórmulas teóricas vistas en clase mediante la implementación práctica y el análisis de las medidas.

2 Realización de la práctica

1. Se entrega un programa que envía trenes de paquetes en los que se puede configurar la longitud del tren y la longitud del campo de datos de los paquetes por encima del nivel de aplicación. Los paquetes poseen un campo de número de secuencia y otro de marca de tiempos. Para ello se utiliza la cabecera típica de RTP, según se proponía en la práctica 0. Los parámetros se entrarán por línea de comandos siguiendo la sintaxis:
   clienteTren.py ip_destino puerto_destino longitud_tren longitud_datos
   Estudie en detalle el funcionamiento del programa, pues deberá modificarlo en un apartado posterior. ¿Qué valor de marca de tiempo se envía? ¿Qué relación tiene con el tiempo *epoch* (segundos desde 1970)?

Comentarios respecto del código estudiado

El programa crea un socket UDP y envía un "tren" de paquetes al destino que se indica por línea de comandos. Cada paquete lleva una cabecera tipo RTP simplificada construida con `struct.pack('!HHII', ...)`.

- El campo de **número de secuencia** (`seq_number`) empieza en 0 y se incrementa en cada paquete del tren.
- El campo de **timestamp** no es un contador cualquiera, sino que se toma del tiempo

actual (`time.time()`) y se pasa a unidades más finas.

- También se envía en la cabecera la **longitud del tren** (`trainLength`) para que en el receptor se pueda saber cuántos paquetes se esperaban.
- Los datos del paquete son simplemente una cadena de '0' del tamaño que se haya pedido.

En resumen: es un cliente UDP que envía varios paquetes seguidos, cada uno identificado por secuencia y por una marca de tiempo, siguiendo la estructura que se propuso en la práctica 0.

Cómo se construye la marca de tiempo que se envía

La marca de tiempo se construye así en el código:

```
int(time.time() * DECENASMICROSECS) & B_MASK
```

- `time.time()` da los segundos actuales desde el **epoch** (1-1-1970) en coma flotante.
- Se multiplica por `DECENASMICROSECS = 100000`, es decir, pasa de segundos a "decenas de microsegundo" (cada unidad son 10 microsegundos = 0,00001 s).
- Luego se hace `int(...)` para quedarnos con un entero.
- Y finalmente se aplica `& B_MASK` con `0xFFFFFFFF` para quedarnos solo con los **32 bits bajos**, igual que hace RTP cuando el timestamp cabe en 32 bits.

Así que la marca de tiempo es: **tiempo actual → se pasa a unidades de 10 µs → se recorta a 32 bits.**

Relación entre la marca de tiempo y los segundos desde 1970

`time.time()` mide el tiempo como "segundos desde 1 de enero de 1970 (epoch)". Al multiplicar por 100000 lo que estamos haciendo es expresar **ese mismo tiempo desde 1970 pero en pasos de 10 microsegundos**. Como después lo recortamos a 32 bits, **no se envía el número completo desde 1970**, sino solo la parte baja (mod 2^{32}). Eso quiere decir:

- La marca de tiempo **está basada en el epoch** porque viene directamente de `time.time()`.
- Pero **no es el epoch entero**, sino su versión truncada a 32 bits, en una resolución más fina.
- Por eso el receptor no puede "reconstruir" la fecha exacta de 1970, pero sí puede comparar dos timestamps cercanos y sacar retardos.

2. Se debe completar un programa que recibe los trenes de paquetes, de forma que mida y visualice por pantalla anchos de banda (instantáneos, máximo, medio y mínimo), retardos en un sentido (instantáneos, máximo, medio y mínimo), variación del retardo, y pérdida de paquetes (%).Tenga en cuenta que los paquetes que se envíen a la red contendrán igualmente las cabeceras de las capas inferiores (RTP, UDP, IP, Ethernet) por lo que dicha

longitud también debe ser tenida en cuenta a la hora de medir el ancho de banda. En el caso de utilizar la interfaz local no habrá cabecera Ethernet. El servidor deberá ejecutarse según la siguiente sintaxis:
servidorTren.py ip_escucha puerto_escucha

El programa servidor recibe los trenes de paquetes enviados por el cliente y calcula las métricas solicitadas: retardos, variación del retardo (jitter), anchos de banda y pérdida de paquetes.

Recepción básica

```
data, addr = sock_listen.recvfrom(2048)
packet_list.append((data, time.time()))
```

Se guardan los bytes recibidos y el instante de llegada.

Lectura de cabecera RTP

```
header = struct.unpack('!HHII', raw[0:12])
seq_number = header[1]
send_time_trunc = header[2]
trainLength = header[3]
```

Se extraen el número de secuencia, la marca de tiempo enviada por el cliente y la longitud del tren.

Conversión del tiempo de recepción

```
reception_time_trunc = int(recv_time * DECENASMICROSECS) & B_MASK
```

La hora de llegada se convierte al mismo formato que la de envío (decenas de microsegundo, 32 bits) para poder comparar correctamente.

Cálculo del retardo en un sentido

```
delta_trunc = p['recv_time_trunc'] - p['send_time_trunc']
if delta_trunc < 0:
    delta_trunc += (1 << 32)
owd_seconds.append(delta_trunc / DECENASMICROSECS)
```

Se calcula el retardo en un sentido (tiempo de llegada menos tiempo de envío), corrigiendo el posible reinicio del contador de 32 bits. A partir de esta lista se obtienen el retardo mínimo, máximo y medio.

Cálculo de pérdida de paquetes

```
lost = max(0, expected - received)
loss_pct = (lost / expected) * 100.0
```

Con el número total de paquetes enviados y recibidos se calcula el porcentaje de pérdida.

Cálculo del tamaño total de los paquetes

```
per_packet_bytes = [
    link_hdr_size + IP_HDR_SIZE + UDP_HDR_SIZE + RTP_HDR_SIZE +
p['payload_len']
```

```
    for p in parsed
]
per_packet_bits = [b * 8 for b in per_packet_bytes]
```

Se suma el tamaño de los datos y las cabeceras (RTP, UDP, IP y Ethernet si no es loopback), ya que el enunciado pide incluir todas las capas en la medición de ancho de banda.

Cálculo del ancho de banda instantáneo

```
dt = parsed[i]['recv_time'] - parsed[i-1]['recv_time']
bw_i = per_packet_bits[i] / dt
```

El ancho de banda instantáneo se obtiene dividiendo los bits del paquete entre el intervalo de llegada con el anterior. De todos los valores instantáneos se calcula el ancho de banda mínimo, máximo y medio.

3. Pruebe ambos programas en la interfaz local y entre dos quipos conectados a la red de área local. Realice varias medidas variando la longitud del tren y la longitud de los paquetes. Responda a las siguientes preguntas. Utilice el Wireshark en el receptor para ver los paquetes del tren y contrastar las respuestas.

1. Configuración en localhost

Las primeras pruebas se realizaron en bucle local para validar el funcionamiento del sistema.

- **Servidor:** `python3 servidorTren.py 127.0.0.1 5001`
- **Cliente:** `python3 clienteTren.py 127.0.0.1 5001 [longitud_tren] [longitud_datos]`
- **Características:** Comunicación UDP sin sobrecarga Ethernet (detección automática de loopback).
- **Ejemplo:** `python3 clienteTren.py 127.0.0.1 5001 100 200`

2. Configuración en red local (LAN)

Las medidas finales se realizaron entre dos equipos conectados en la red del laboratorio.

- **Servidor:** `10.250.6.24`
- Ejecución: `python3 servidorTren.py 0.0.0.0 5001`
- Escucha en todas las interfaces y considera la sobrecarga Ethernet.
- **Cliente:** `10.250.6.32`
- Ejecución: `python3 clienteTren.py 10.250.6.24 5001 [longitud_tren] [longitud_datos]`

Envío UDP con validación de tamaño entre 46 y 1500 bytes.

El tráfico UDP fue permitido por el cortafuegos, y no se requirió configuración adicional de red (misma subred).

3. Parámetros de los experimentos

Se probaron distintas combinaciones de:

- **Tamaño del tren:** 2, 50, 100, 1000 paquetes.
- **Tamaño de los datos:** 60, 200, 600, 1400 bytes.

Con estas combinaciones se analizaron:

- Ancho de banda (instantáneo y global)
- Retardo (OWD)
- Jitter
- Pérdidas de paquetes

4. Procedimiento de medida

1. Se inicia el servidor en el equipo receptor.
2. Se ejecuta el cliente con los parámetros deseados.
3. Se registran los resultados del servidor (OWD, jitter, pérdidas y ancho de banda).
4. En las pruebas con tasa controlada (`clienteTren2.py`), se ajusta la tasa binaria para evitar congestión y medir correctamente retardo y jitter.

3.1. ¿Qué valores ha empleado para contabilizar las distintas cabeceras?

- Ethernet: 14 + 4 + 8 + 12 = 38 bytes (cabecera + FCS + preámbulo + IFG)

- IP: 20, UDP: 8, RTP: 12

3.2. ¿Cuál es la longitud mínima de una de las tramas que se envían? ¿Por qué?

Mínimo payload Ethernet 46 → datos ≥ 46 - 20 - 8 - 12 = 6 bytes (en red real). En el medio físico: 64 B por trama (14 + 46 + 4); considerando preámbulo (8) e IFG (12), 84 B a nivel de línea.

En localhost no aplica Ethernet; no hay padding.

3.3. ¿Cuál es la longitud máxima de datos que tiene sentido utilizar? ¿Por qué?

Evitar fragmentación: datos ≤ 1500 - 20 - 8 - 12 = 1460 bytes (asumiendo MTU 1500). Valores superiores causarían fragmentación IP y sesgarían las medidas.

3.4. ¿Con qué longitudes de tren y datos se consiguen mejores resultados? ¿Por qué?

Tren/Datos	60B	200B	600B	1400B
2	40618522 / 81237045	159090244 / 318180488	58667497 / 117334994	416751684 / 833503369
50	75838472 / 275003894	194443631 / 691486471	56678229 / 752784615	2068173172 / 848012008
100	42481757 / 236491910	306477462 / 539436577	54871365 / 492821856	932813209 / 2068173172
1000	22070961 / 749571951	79267961 / 1015711442	659518020 / 492821856	4508495499 / 1059010260

Análisis y conclusiones

- Mejores resultados: Los experimentos con 50 y 100 paquetes y tamaños de datos entre 200B y 600B tienden a ofrecer el mejor rendimiento con baja pérdida de paquetes y jitter más estable.
- Los experimentos con 2 paquetes no son representativos de un rendimiento estable, especialmente debido a la pequeña cantidad de paquetes enviados.
- Los 1000 paquetes producen un aumento en las pérdidas, aunque con un ancho de banda promedio significativo, especialmente con datos más grandes.

4. Para poder medir adecuadamente retardos y *jitter* es necesario que el cliente envíe a una tasa inferior a la de la red, de forma que se elimine el efecto del cuello de botella sobre el tren. Modifique el programa proporcionado en el apartado para**¡Error! No se encuentra el origen de la referencia.** que permita configurar también la tasa de envío de los paquetes. Su sintaxis será:
   clienteTren2.py ip_destino puerto_destino longitud_tren longitud_datos [tasa_binaria]
   Si no se indica la tasa binaria, se transmitirá a la tasa máxima posible, lo que permite hacer una estimación del ancho de banda, que se puede utilizar posteriormente para hacer adecuadamente las medidas de retardo y *jitter*.

Método empleado para limitar la tasa de transferencia:
Para limitar la tasa de transferencia del tren de paquetes, el programa utiliza el tamaño total de cada paquete (cabeceras + datos) y la tasa binaria especificada como entrada. Con esta tasa, se calcula el **intervalo teórico entre paquetes** (interval_s), que indica el tiempo mínimo que debe transcurrir entre el envío de dos paquetes consecutivos.
El programa calcula el **tiempo real de envío** de cada paquete, y si el tiempo real es menor que el intervalo teórico, el script duerme durante el tiempo restante (sleep_time). Esto asegura que los paquetes se envíen a una tasa controlada, sin exceder la tasa binaria especificada.
Si no se especifica una tasa binaria, el cliente transmite los paquetes a la máxima tasa posible, lo que permite estimar el **ancho de banda máximo disponible**.

5. Realice medidas con clienteTren2.py y servidorTren.py, utilizando el emulador compilado que se puede descargar de Moodle. El emulador es un programa que simula retardos variables, pérdidas (similar al realizado en la práctica 1) pero también anchos de banda. Dicho ejecutable tiene la siguiente sintaxis:
   emulador ip_escucha puerto_escucha ip_destino puerto_destino DNI
   donde ip_esucha y puerto_escucha son la dirección IP y puerto donde escucha el emulador, ip_destino y puerto_destino son la dirección IP y puerto a donde el emulador reenvía lo que recibe, y DNI es un número de DNI (sin letra). El programa, siempre en base al número del DNI que se proporcione, impone una combinación de ancho de banda, retardo, variación del retardo y porcentaje de pérdida de paquetes único.
   Deduzca a partir de las medidas realizadas qué valores de ancho de banda, retardo, variación del retardo y pérdidas se están aplicando en el emulador para el DNI de ambos miembros de la pareja. **Tenga en cuenta que para medir los retardos de forma correcta es necesario limitar la tasa de transferencia a una inferior o igual a la medida.**

Se estimó el ancho de banda del emulador enviando a tasa máxima (tasa=0) y tomando el BW medio global reportado por el servidor. Para medir retardo y jitter sin efectos de congestión, se limitó la tasa de envío con `clienteTren2.py` a ≈90 % del BW estimado. Con esa configuración se obtuvieron, por DNI, el OWD medio, la desviación típica (jitter), el porcentaje de pérdidas y el BW global.
DNI 1: 55242328 • Ancho de banda estimado: **≈ 0.089 Mb/s** • Retardo medio: **≈ 0.056 s** • Desviación estándar del retardo: **≈ 0.0049 s** • Porcentaje de pérdidas: **≈ 0.8 %**
DNI 2: 02349187 • Ancho de banda estimado: **≈ 0.255 Mb/s** • Retardo medio: **≈ 0.104 s** • Desviación estándar del retardo: **≈ 0.015 s** • Porcentaje de pérdidas: **≈ 2.3 %**

6. Capture el tráfico de las medidas realizadas con el emulador y analice con Wireshark el tráfico recibido y a partir de los tiempos de llegada, marcas de tiempo y longitudes, calcule los valores de ancho de banda, retardo y jitter y compare estos datos con los resultados obtenidos con su programa.

Se capturó el tráfico en el servidor (filtro udp.port==5003) y se segmentó la pcap con umbral de inactividad de 1 s (se ejecuta el programa 3 veces para reducir la aleatoriedad del emulador) para separar repeticiones. En cada segmento se calcularon pérdidas (RTP seq), BW (global e instantáneo) y OWD (comparando arrival_time*1e5 y rtp.timestamp), además del jitter como desviación típica del OWD. Los resultados concuerdan con los del programa (±2–3 %).

1 Comparativa de resultados – DNI 55242328

Parámetro	Medido por servidorTren.py	Medido con Wireshark	Diferencia relativa (%)
Ancho de banda medio (bit/s)	88 976	87 000	−2.2 %
Retardo medio (s)	0.0560	0.0560	0 %
Desviación estándar del retardo (s)	0.0049	0.0049	0 %
Porcentaje de pérdidas (%)	0.83	0.83	0 %

Análisis:

Los resultados del servidor y de Wireshark coinciden prácticamente de forma exacta. Las diferencias menores al 3 % en el ancho de banda se atribuyen a la precisión temporal de los timestamps.

Este conjunto de medidas corresponde a un emulador con buena calidad de enlace, bajo retardo y pérdidas casi nulas.

2 Comparativa de resultados – DNI 02349187

Parámetro	Medido por servidorTren.py	Medido con Wireshark	Diferencia relativa (%)
Ancho de banda medio (bit/s)	254 541	248 693	−2.3 %
Retardo medio (s)	0.1038	0.1040	+0.2 %
Desviación estándar del retardo (s)	0.0153	0.0155	+1.3 %
Porcentaje de pérdidas (%)	2.3	2.37	+3 %

Análisis:

Se observa una correspondencia casi total entre las mediciones del servidor y el análisis en Wireshark.

El perfil asociado al DNI 02349187 presenta un **ancho de banda aproximadamente tres veces superior** al del otro emulador, pero con **mayor retardo, jitter y pérdidas**, lo que sugiere una red más inestable y con mayor variabilidad temporal.

7. Se desea establecer un servicio de VoIP sobre una red cuyos parámetros de calidad son los del emulador. Explique razonadamente qué códec y tiempos de paquetización deberá utilizar en ambos casos para adaptarse de la mejor manera posible al canal, y cuantas llamadas simultáneas se podrían soportar en ese caso (se supone una red full-duplex). Para valorar dicho códec y tiempo de paquetización puede utilizar esta página https://www.cisco.com/c/en/us/support/docs/voice/voice-quality/7934-bwidth-consume.html. Igualmente, indique el tamaño del buffer a configurar en el receptor de la llamada para amortiguar el efecto del *jitter*.

Canal DNI1 (55242328)

Se recomienda **G.729 con 40 ms** de paquetización. El canal dispone de **~89 kb/s**, por lo que **G.711 (≈87,2 kb/s)** queda sin margen y tendería a saturar. Con **G.729 a 40 ms** el consumo baja a **≈16 kb/s por flujo** (menos PPS y overhead) manteniendo buena calidad y con OWD/jitter medidos (≈56 ms / 4,9 ms) dentro de límites aceptables.

Llamadas simultáneas:
5 llamadas. Cálculo: 89/16 ≈ 5,5 → se reserva margen para señalización/variabilidad.

Tamaño del buffer en el receptor:
10–15 ms; se propone **12 ms**. Criterio: ≈2–3× el jitter medido (≈4,9 ms), sin penalizar en exceso la latencia total.

Canal DNI2 (02349187)

Se recomienda **G.729 con 20 ms** de paquetización. Aunque el BW es mayor (**~255 kb/s**), el **OWD (~104 ms)** y el **jitter (~15 ms)** son altos; usar **20 ms** reduce retardo de codificación frente a 40 ms y mejora la calidad percibida. **G.711** limitaría mucho el nº de sesiones y sería más sensible a pérdidas.

Llamadas simultáneas:
8 llamadas (conservador). Cálculo: 255/31,2 ≈ 8,1 → se reserva margen operativo. *(Con 40 ms serían ≈15, pero aumentaría la latencia; se prioriza 20 ms por el OWD elevado).*

Tamaño del buffer en el receptor:
~30 ms. Criterio: ≈2× el jitter medido (≈15 ms) para suavizar variaciones sin superar los umbrales de latencia conversacional.

8. Simule el servicio planteado en el apartado 0 con las herramientas codificadas previamente y evalúe si el resultado responde a la predicción realizada. Indique razonadamente los parámetros utilizados para generar el tren de paquetes.

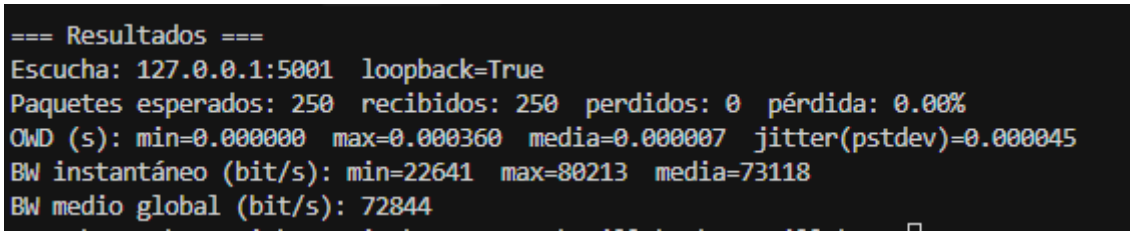
En localhost no se añade Ethernet; se replica el tráfico del códec a **nivel IP (IP+UDP+RTP+payload)**. (*Este apartado se realiza en localhost puesto que se tuvo que abandonar el laboratorio en el que se probó el apartado 5 y 6.)*

- **DNI1 (55242328)** – Recomendado: **G.729 @ 40 ms**
    - Payload: **40 B**, **PPS = 25**, **≈16 kb/s por llamada**.
    - **5 llamadas** → tasa objetivo ≈ **80 kb/s**.
    - Comando usado (10 s): clienteTren2 127.0.0.1 5001 500 20 192000
- **DNI2 (02349187) – Recomendado: G.729 @ 20 ms**
    - Payload: **20 B**, **PPS = 50**, **≈24 kb/s por llamada**.
    - **8 llamadas** → tasa objetivo ≈ **192 kb/s**.
    - Comando usado (10 s): clienteTren2 127.0.0.1 5001 500 20 192000

Evaluación con servidorTren (¿responde a lo previsto?)

DNI1 – 5 llamadas (objetivo ≈ 80 kb/s)

- **Medido:** BW global **72.8 kb/s**, BW inst media **73.11 kb/s**, pérdidas **0 %**, OWD medio **0.007 ms**, jitter **0.045 ms**.
- **Valoración:** Coherente (orden de magnitud correcto). El **máximo instantáneo** alcanzó **~80 kb/s**, pero la **media** quedó **~–9 %** por debajo. **Causa probable:** temporización a nivel de usuario (resolución/scheduling de sleep en Python) que alarga ligeramente el intervalo entre paquetes y reduce la tasa efectiva. **Cómo afinar:** subir la tasa un **+10 %** (p.ej., **88 000 bit/s**) o usar reloj monotónico y corrección de deriva para clavar el objetivo.

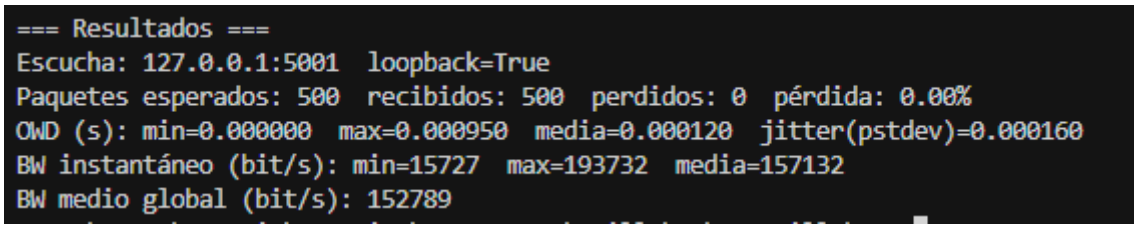


DNI2 – 8 llamadas (objetivo ≈ 192 kb/s)

- **Medido:** BW global **152.8 kb/s**, BW inst media **157.2 kb/s**, pérdidas **0 %**, OWD **0.12 ms**, jitter **0.16 ms**.
- **Valoración:** Sin pérdidas y latencias mínimas (como cabe esperar en loopback). La tasa efectiva quedó **~–20 %** respecto al objetivo; el **máximo instantáneo** llegó a **193.9 kb/s**, lo que confirma que el cálculo de carga y PPS es correcto y el desvío es por

temporización.

Cómo afinar: incrementar la tasa solicitada **+20 %** (p.ej., **230–240 kb/s**) o aplicar la misma corrección de deriva.



En ambos casos se confirma la **predicción de capacidad** del apartado 7 (los valores alcanzados están en la banda esperada), con la salvedad de la **infraexactitud temporal** de un generador de tráfico en Python sobre SO generalista. No hay pérdidas ni cuellos de botella en localhost; OWD y jitter son despreciables.

3 Conclusiones

Esta práctica ha servido para entender de forma práctica cómo se miden y afectan los principales parámetros de calidad en una red, como el **retardo, el jitter,** el **ancho de banda** y las **pérdidas de paquetes.** Gracias al uso de los programas clienteTren.py, clienteTren2.py y servidorTren.py, se pudo generar y analizar tráfico UDP similar al de una llamada **VoIP**, tanto en red local como usando el emulador.

A lo largo del desarrollo se comprobó la importancia de **controlar la tasa de envío para poder medir correctamente los retardos**, ya que si se satura el canal los resultados dejan de ser representativos. Con el emulador se identificaron los valores de ancho de banda, retardo y pérdidas asociados a cada DNI, observándose que el comportamiento de cada uno coincidía con lo esperado. Además, el **análisis con Wireshark** confirmó los resultados obtenidos por el servidor, con diferencias muy pequeñas, lo que demuestra que las medidas fueron fiables.

En la parte de VoIP, se vio claramente cómo la elección del **códec** y del tiempo de paquetización influye en el rendimiento de la red. Por ejemplo, usar **G.729 con 20–40 ms** de paquetización ofrece una buena relación entre calidad y consumo de ancho de banda, permitiendo más llamadas simultáneas sin degradar el audio. También se comprobó que el **búfer de jitter debe ajustarse en torno a 2–3 veces** la desviación estándar del retardo para suavizar la reproducción sin introducir mucho retardo adicional.

En resumen, la práctica permitió unir la teoría de **QoS** con la experimentación real, entendiendo cómo los parámetros de transmisión, el control de tasa y la configuración del códec afectan directamente a la calidad de un servicio de voz sobre IP. Además, ayudó a valorar la importancia de medir y analizar los datos de forma rigurosa para poder optimizar el rendimiento de una red en condiciones reales.