SISTEMAS DISTRIBUIDOS MEMORIA

**Autores:** Vladimir Iarunichev y Francisco Rebolo Cabo

Índice

[Práctica 1 3](#_Toc210323622)

[Parte 1: 3](#_Toc210323623)

[EJ-1: Servidor y Cliente UDP con parámetros por defecto 3](#_Toc210323624)

[EJ-2: UDP con pérdida simulada y numeración de mensajes 5](#_Toc210323625)

[EJ-3: UDP con confirmación y detección de pérdida 6](#_Toc210323626)

[EJ-4: Cliente UDP con reintentos y timeout progresivo 8](#_Toc210323627)

[EJ-5: UDP con control de duplicados y reintentos inteligentes 10](#_Toc210323628)

[EJ-6: Descubrimiento de servidores UDP por broadcast 13](#_Toc210323629)

[EJ-7: Descubrimiento UDP por broadcast en Docker 14](#_Toc210323630)

[Parte 2: 17](#_Toc210323631)

[EJ-1: Servidor y cliente TCP con envío secuencial 17](#_Toc210323632)

[EJ-2: CP con recvall y sendall para transmisión fiable 17](#_Toc210323633)

[EXPERIMENTO: TCP: efecto del tamaño de mensaje y sincronización 17](#_Toc210323634)

[EJ-4 17](#_Toc210323635)

[EJ-5 17](#_Toc210323636)

[EJ-6 17](#_Toc210323637)

[EJ-7 17](#_Toc210323638)

# Práctica 1

## Parte 1:

### EJ-1: Servidor y Cliente UDP con parámetros por defecto

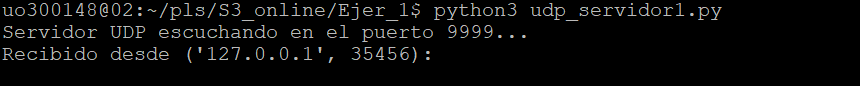
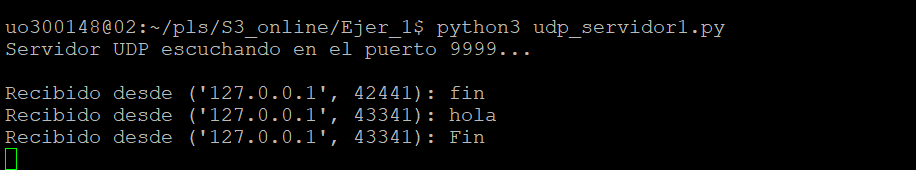


foto ejecutando la prueba en la ventana de la maquina virtual





**1. Introducción**

En esta primera práctica hemos trabajado con **sockets UDP** en Python. El objetivo del ejercicio es comprender el funcionamiento básico del protocolo UDP, implementando un servidor y un cliente sencillos, y verificando la comunicación entre ambos. Además, se realizaron pruebas iniciales con la herramienta netcat para confirmar el envío y recepción de datagramas.

**2. Resolución del ejercicio**

**2.1. Servidor UDP**

Se implementó un servidor en Python (udp\_servidor1.py) que:

* Escucha en un puerto (por defecto, 9999).
* Permanece en espera en un bucle infinito.
* Muestra en pantalla los mensajes recibidos y la dirección IP de origen.

El código completo se encuentra en el repositorio Git del grupo.

**3. Actividades y experimentos**

* Se verificó el funcionamiento del servidor con netcat.
* Se probó la comunicación **cliente–servidor** en dos terminales distintas de la misma máquina virtual.
* Se comprobó que el servidor soporta múltiples mensajes hasta que el cliente finaliza.

**Observación:** dado que UDP es un protocolo no orientado a conexión, no se garantiza la entrega, pero en las pruebas realizadas en entorno local todos los datagramas fueron recibidos correctamente.

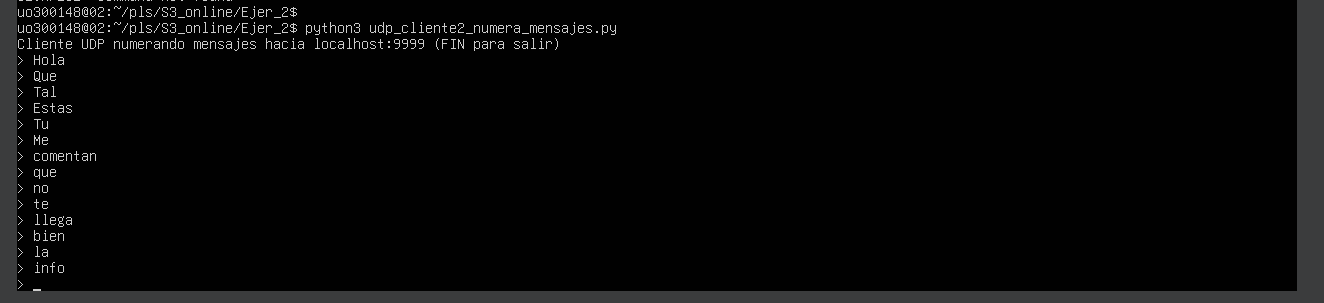
**4. Reflexiones y conclusiones**

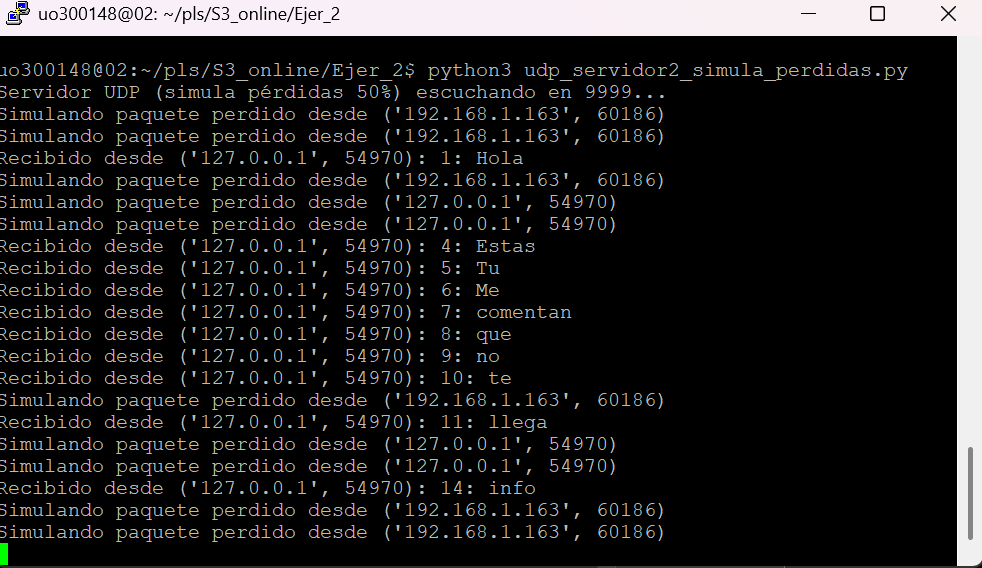
Este primer ejercicio nos permitió:

* Entender la diferencia entre TCP y UDP, especialmente en que UDP no establece una conexión persistente.
* Practicar la creación de sockets y el envío/recepción de datagramas en Python.
* Familiarizarnos con herramientas básicas de diagnóstico como netcat.

Consideramos que fue un ejercicio útil para sentar las bases de las siguientes prácticas, donde la complejidad de las aplicaciones irá aumentando.

### EJ-2: UDP con pérdida simulada y numeración de mensajes





**1. Introducción**

En este ejercicio se amplió el programa del **Ejercicio 1** para simular el comportamiento no confiable de UDP. El objetivo fue implementar un servidor que **pierde aleatoriamente algunos datagramas** con una probabilidad definida, y un cliente que **numera los mensajes** enviados para poder detectar fácilmente dichas pérdidas en la secuencia.

**2. Resolución del ejercicio**

**2.1. Servidor UDP con pérdidas simuladas**

Se creó el archivo udp\_servidor2\_simula\_perdidas.py a partir del servidor anterior.

* Tras recibir un datagrama, el servidor decide aleatoriamente (50% por defecto) si simula una pérdida o lo procesa.
* En caso de pérdida imprime en pantalla: Simulando paquete perdido desde (IP, puerto)
* En caso contrario, muestra el contenido recibido y la dirección de origen.
* Se añadió la posibilidad de configurar la probabilidad de pérdida como argumento al ejecutar el programa, y un resumen de estadísticas al finalizar con Ctrl+C.

**2.2. Cliente UDP con numeración de mensajes**

Se creó el archivo udp\_cliente2\_numera\_mensajes.py, que amplía el cliente anterior:

* Cada datagrama lleva un número incremental al inicio, en formato "n: mensaje".
* Esto permite observar fácilmente en el servidor los **saltos en la secuencia** cuando se pierden paquetes.
* El programa finaliza al introducir FIN.

**3. Resultados observados**

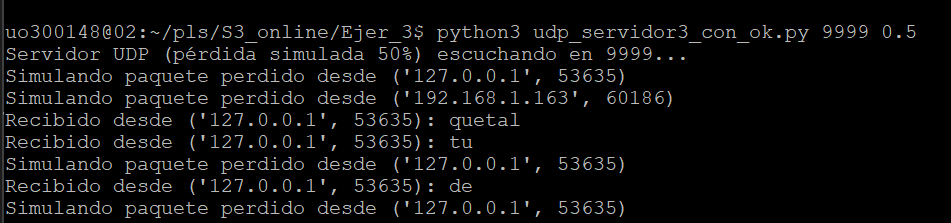
* El servidor mostró correctamente tanto los mensajes recibidos como los que se marcaron como “paquete perdido”.
* En la secuencia de números enviados por el cliente se observaron **saltos** (ejemplo: se reciben 1, 3, 5), lo cual indica que los datagramas 2 y 4 se perdieron en la simulación.
* Al interrumpir el servidor con Ctrl+C se obtuvo un resumen con el porcentaje real de pérdidas, que coincidió aproximadamente con la probabilidad configurada.

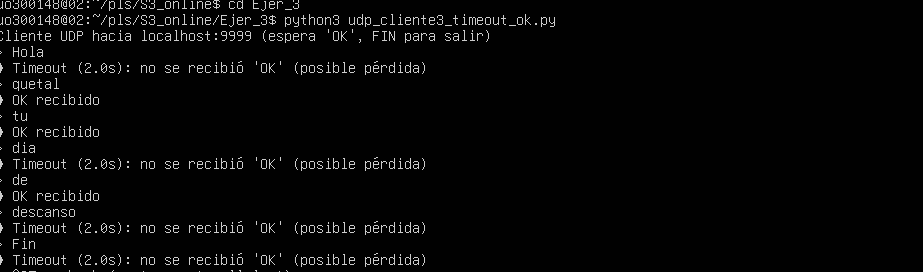
**4. Reflexiones y conclusiones**

Este ejercicio permitió comprender mejor:

* Cómo se pueden **simular fallos de red** en un entorno controlado.
* Que UDP no garantiza la entrega, y cómo la pérdida de datagramas afecta directamente a la aplicación.
* La importancia de numerar los mensajes para detectar pérdidas, técnica usada en protocolos de nivel superior como TCP.

### EJ-3: UDP con confirmación y detección de pérdida





**1. Introducción**

En este ejercicio se mejoró la comunicación cliente-servidor UDP para **detectar la pérdida de paquetes**.  
El objetivo es que el **servidor confirme la recepción** de cada datagrama mediante un mensaje de reconocimiento "OK".  
De esta forma, el cliente puede saber si su mensaje llegó correctamente o si se perdió, aplicando además un **timeout** para no quedar bloqueado indefinidamente en la espera de respuesta.

**2. Resolución del ejercicio**

**2.1. Servidor UDP con confirmación**

Se creó el archivo udp\_servidor3\_con\_ok.py, basado en el servidor del ejercicio anterior.

* El servidor sigue simulando pérdida con una probabilidad configurable (50% por defecto).
* Si un datagrama “se pierde”, se muestra en pantalla el mensaje Simulando paquete perdido y **no se responde**.
* Si el datagrama es aceptado, se muestra por pantalla y se responde al cliente con "OK".

**2.2. Cliente UDP con espera de "OK"**

Se creó el archivo udp\_cliente3\_timeout\_ok.py.

* Envía mensajes al servidor y, tras cada envío, ejecuta un recvfrom() esperando "OK".
* Se configuró un **timeout de 2 segundos**: si no llega ninguna respuesta en ese tiempo, el cliente asume que hubo pérdida (ya sea del datagrama o de la respuesta).
* El programa informa al usuario con:
  + ✓ OK recibido si llega la confirmación.
  + ⏱ Timeout: no se recibió 'OK' si se agota el tiempo de espera.

**3. Resultados observados**

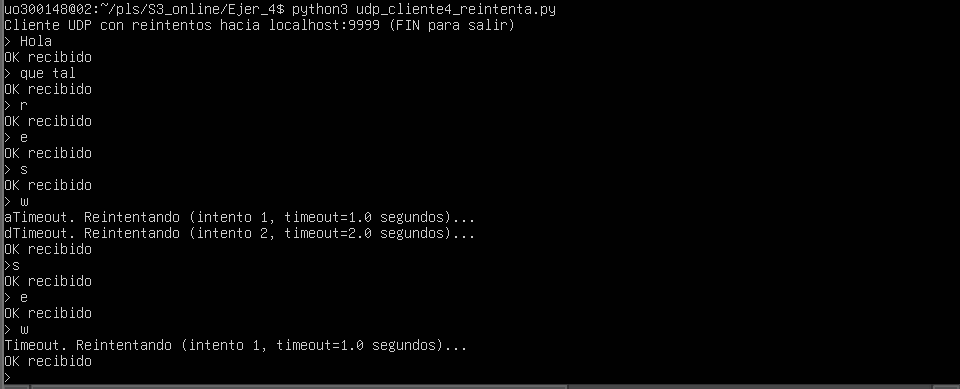
* Cuando no hay pérdida, el cliente recibe el "OK" del servidor en menos de 2 segundos.
* Cuando el servidor “pierde” un datagrama, el cliente muestra un timeout y sabe que el envío no fue confirmado.
* También se observó que, si el "OK" se pierde, el cliente se comporta igual que si se hubiese perdido el mensaje original, lo que refleja un escenario real de red.

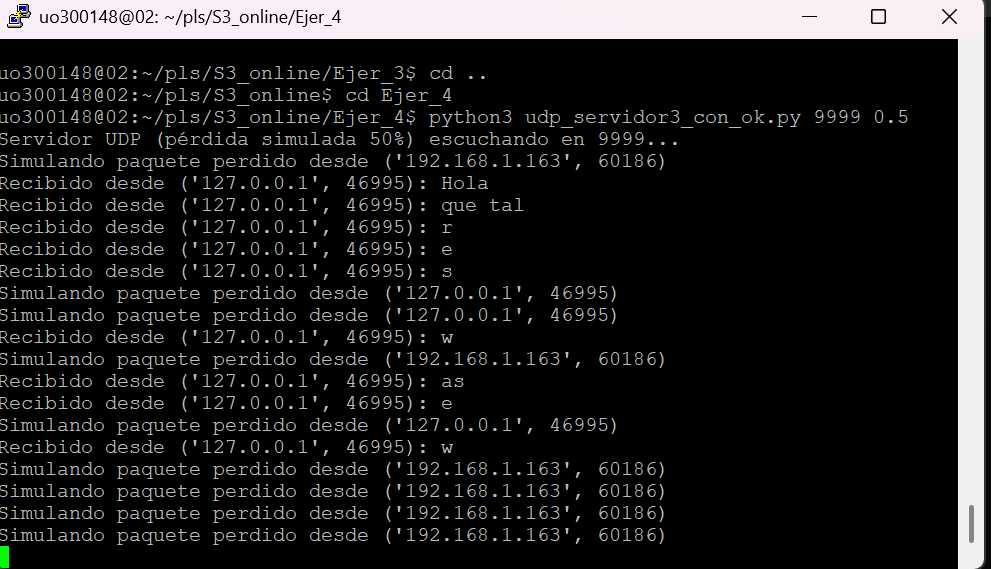
**4. Reflexiones y conclusiones**

Con este ejercicio se comprendió que:

* **UDP no garantiza fiabilidad**, por lo que protocolos superiores deben implementar mecanismos de confirmación.
* Añadir un **reconocimiento ("OK")** y un **timeout** permite al cliente detectar pérdidas y no quedar bloqueado.
* Este patrón es la base de cómo protocolos como **TCP** implementan confirmaciones, retransmisiones y gestión de errores.

### EJ-4: Cliente UDP con reintentos y timeout progresivo





**1. Introducción**

En este ejercicio se amplió la funcionalidad del cliente UDP para que no solo detecte pérdidas de mensajes, sino que también **reintente el envío** en caso de no recibir confirmación. El objetivo es acercarse al funcionamiento de protocolos fiables como TCP, implementando un mecanismo básico de retransmisión con incremento del tiempo de espera entre reintentos.

**2. Resolución del ejercicio**

**2.1. Servidor empleado**

Para esta práctica se utilizó el servidor del Ejercicio 3 (udp\_servidor3\_con\_ok.py), que responde con un mensaje "OK" cuando recibe correctamente un datagrama.

**2.2. Cliente con reintentos**

Se creó el archivo udp\_cliente4\_reintenta.py, partiendo del cliente anterior. Sus características principales son:

* Envía el mensaje al servidor y espera recibir un "OK".
* Si no recibe respuesta en un tiempo (timeout), vuelve a enviar el mismo mensaje.
* El tiempo de espera comienza en **0.5 segundos** y se **duplica en cada reintento** (0.5 → 1.0 → 2.0).
* Si el tiempo de espera supera los **2 segundos**, el cliente asume que el servidor puede estar caído y detiene los reintentos, mostrando un mensaje de aviso.

**3. Resultados observados**

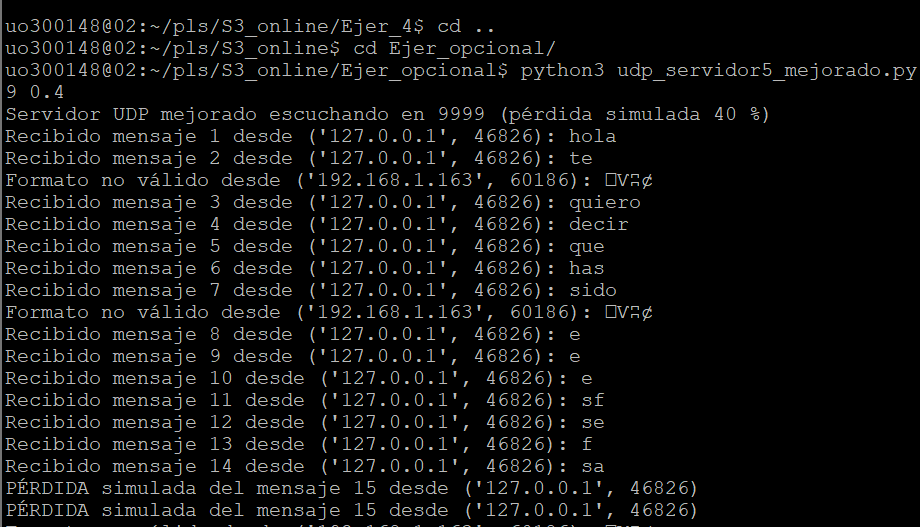
* Cuando el servidor recibe y responde, el cliente muestra el mensaje:
* OK recibido
* Cuando no se recibe el "OK", el cliente repite el envío, aumentando progresivamente el tiempo de espera.
* Tras varios intentos, si el tiempo de espera supera los 2 segundos, el cliente detiene los reintentos e informa:
* Puede que el servidor esté caído. Inténtelo más tarde.
* De este modo, el cliente puede **detectar fallos y actuar en consecuencia**, sin quedar bloqueado indefinidamente.

4. Reflexiones y conclusiones

Este ejercicio mostró cómo se pueden implementar mecanismos de **reintento y control de tiempo de espera** en protocolos no fiables como UDP.  
El cliente ahora no solo detecta pérdidas, sino que también intenta resolverlas reenviando el paquete, y toma una decisión final si el servidor no responde.  
Este comportamiento reproduce de manera simplificada el funcionamiento de protocolos confiables que se construyen sobre UDP o que forman parte de TCP/IP.

### EJ-5: UDP con control de duplicados y reintentos inteligentes





**1. Introducción**

En este ejercicio opcional se buscó mejorar la fiabilidad del protocolo implementado en los ejercicios anteriores. El problema principal detectado era que la confirmación "OK" no incluía ninguna referencia al mensaje original, lo que podía causar confusiones en caso de reintentos, pérdidas o llegada desordenada de paquetes.

La mejora implementada consistió en **añadir identificadores únicos a cada datagrama** y hacer que el servidor devolviera una confirmación "OK <id>", asociada explícitamente al mensaje recibido. De esta forma, el cliente puede distinguir con claridad qué mensaje fue confirmado y descartar respuestas erróneas o duplicadas.

Además, el servidor se modificó para **simular pérdidas con probabilidad configurable**, lo que permitió probar el sistema de reintentos en condiciones de red adversas.

**2. Resolución del ejercicio**

**2.1. Servidor mejorado**

Se creó el archivo udp\_servidor5\_mejorado.py.

* Escucha en el puerto indicado (por defecto 9999).
* Para cada mensaje recibido, comprueba si se pierde o no según una probabilidad definida al ejecutar el programa.
* Si el datagrama llega, el servidor lo procesa y responde con "OK <id>".
* Si el mensaje ya había sido recibido (reintento), el servidor no lo procesa de nuevo, pero reenvía la confirmación para evitar acciones duplicadas.

**2.2. Cliente mejorado**

Se creó el archivo udp\_cliente5\_mejorado.py.

* Asigna a cada mensaje un identificador secuencial (1: mensaje, 2: mensaje, etc.).
* Envía el datagrama y espera la confirmación "OK <id>" correspondiente.
* En caso de timeout, reenvía el mismo datagrama aumentando el tiempo de espera (0.5 → 1.0 → 2.0 segundos).
* Si tras los reintentos no llega la confirmación, informa que el servidor puede estar caído o que hubo pérdidas persistentes.
* Solo incrementa el número de mensaje cuando se recibe la confirmación correcta.

**3. Resultados observados**

* El cliente solo confirma mensajes cuando recibe el "OK <id>" correcto.
* En caso de pérdida, el cliente repite el envío del mismo mensaje hasta recibir la confirmación o hasta que los reintentos se agotan.
* El servidor detecta duplicados y no repite la acción asociada al mensaje, enviando únicamente la confirmación correspondiente.
* Con esta mejora se evitó el problema de confundir confirmaciones de mensajes distintos y se redujo el riesgo de duplicados procesados.

**4. Reflexiones y conclusiones**

Este ejercicio opcional permitió implementar un mecanismo más robusto de confirmación en UDP:

* Los **identificadores** permiten al cliente y al servidor asociar confirmaciones a mensajes concretos.
* El **control de duplicados** en el servidor evita que se procese dos veces un mismo datagrama reenviado.
* La **simulación de pérdidas** fue útil para verificar el correcto funcionamiento de los reintentos.

En conjunto, estas mejoras hacen que el protocolo se acerque más al comportamiento fiable de TCP, sin dejar de estar implementado sobre UDP.

### EJ-6: Descubrimiento de servidores UDP por broadcast

**Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.**

**Objetivo**

Implementar un sistema de descubrimiento automático de servicios en red local usando UDP broadcast, permitiendo que un cliente encuentre servidores disponibles sin conocer sus direcciones IP específicas.

**Componentes Implementados**

**Servidor de Broadcast**

* Escucha en puerto configurable (12345 por defecto)
* Responde "HOLA" a cualquier mensaje "SERVICIO" recibido
* Permite que múltiples servidores coexistan en la misma red

**Cliente de Broadcast**

* Envía mensaje "SERVICIO" por broadcast (255.255.255.255)
* Recoge respuestas de todos los servidores disponibles
* Selecciona automáticamente el primer servidor que responde
* Implementa timeout de 3 segundos para evitar bloqueos

**Protocolo de Comunicación**

1. **Descubrimiento**: Cliente → Broadcast "SERVICIO"
2. **Respuesta**: Servidores → Cliente "HOLA"
3. **Confirmación**: Cliente → Servidor seleccionado "HOLA cliente"

**Resultados Obtenidos**

* Descubrimiento exitoso de múltiples servidores en red local
* Comunicación funcional entre máquina virtual y sistema anfitrión
* Tiempos de respuesta inferiores a 1 segundo
* Manejo robusto de escenarios sin servidores disponibles

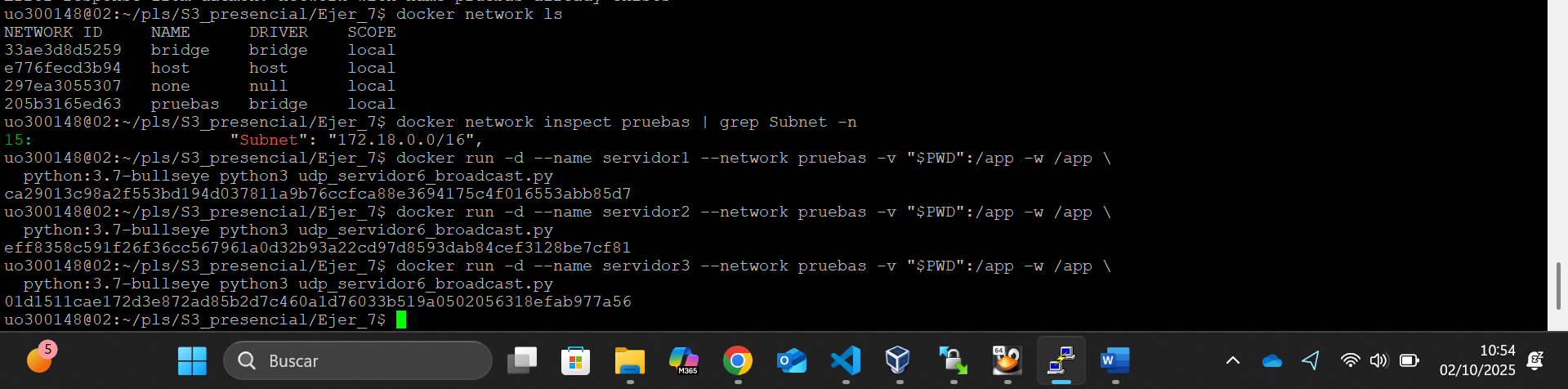
**Aplicaciones Prácticas**

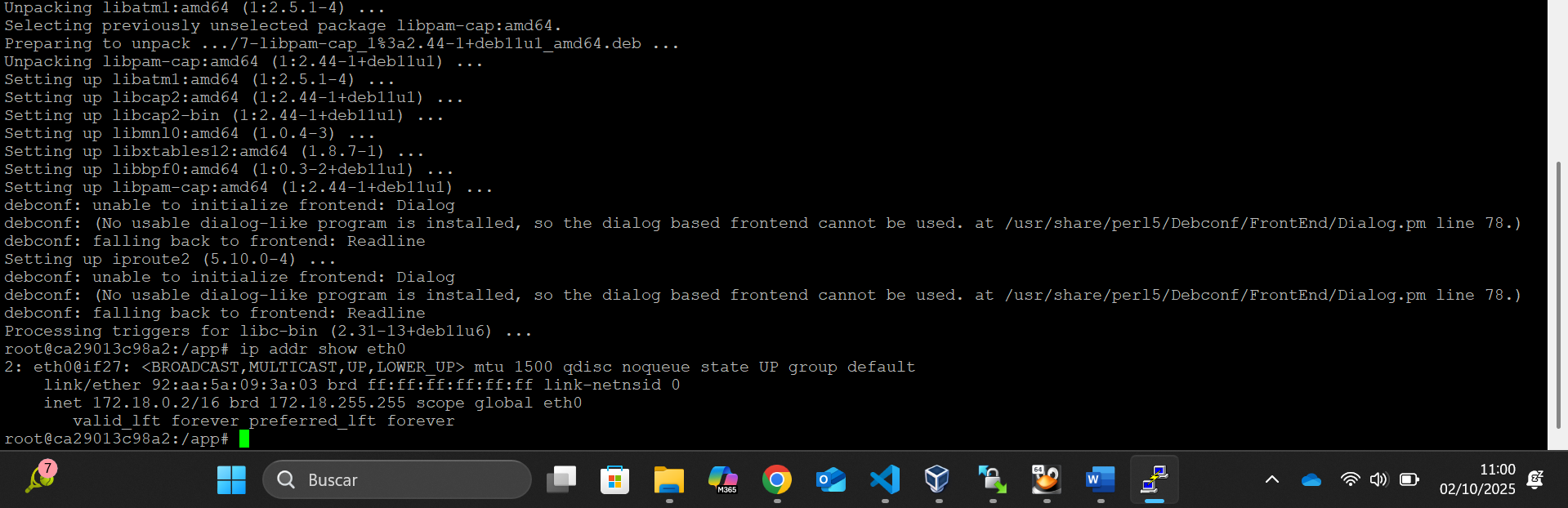
* Sistemas de monitorización distribuida
* Juegos en red local
* Servicios de impresión y recursos compartidos
* Entornos de cluster y computación distribuida

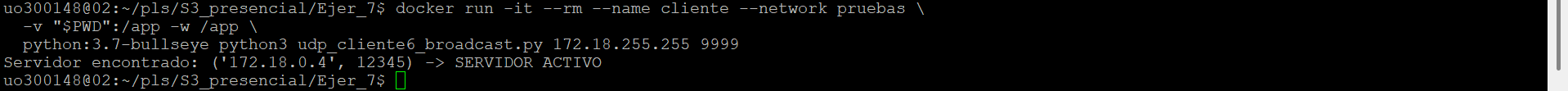
**Conclusión**

El ejercicio demostró la efectividad del broadcast UDP para descubrimiento automático de servicios en red local, proporcionando una solución escalable y de fácil implementación para sistemas distribuidos.

### EJ-7: Descubrimiento UDP por broadcast en Docker







sde borra los servers el cliente(como su ejecucion termino se borro antes el solo) y la red prueba


**Objetivo**

Desplegar tres servidores UDP que escuchan mensajes de descubrimiento por **broadcast** dentro de una subred Docker aislada, y ejecutar un cliente que difunde a la dirección de broadcast, detecta a los servidores y se comunica con el primero que responde.

**Entorno y preparación**

* Carpeta de trabajo con los programas del ejercicio: servidor de broadcast y cliente.
* Imagen de Python basada en Debian (variante “bullseye”), para disponer de utilidades de sistema.
* Red Docker llamada **pruebas** (si ya existía, se reutilizó).

**Procedimiento realizado**

1. **Red de trabajo**  
   Se utilizó la red Docker pruebas para aislar el laboratorio y permitir la resolución de nombres entre contenedores.
2. **Lanzamiento de los servidores**  
   Se iniciaron **tres contenedores** (servidor1, servidor2 y servidor3), todos conectados a la red pruebas, montando la carpeta local como /app y ejecutando el script del servidor en primer plano del contenedor.
3. **Comprobación inicial**  
   Se verificó que los tres servidores estuvieran “Up” y asociados a la red pruebas.  
   (Captura recomendada: listado de contenedores mostrando los tres servidores activos y la red.)
4. **Obtención de la subred y cálculo de broadcast**  
   Se inspeccionó la red pruebas para conocer su subred. El sistema devolvió **172.18.0.0/16**.  
   A partir de esa máscara (/16), la dirección de broadcast correspondiente es **172.18.255.255**.  
   (Captura recomendada: inspección de la red mostrando la subred.)
5. **Ejecución del cliente**  
   Se lanzó un contenedor en la misma red pruebas con el script del cliente, indicándole como destino la **IP de broadcast 172.18.255.255** y el puerto del ejercicio. El cliente difundió su mensaje de descubrimiento y recogió las respuestas.
6. **Resultado de la detección**  
   El cliente informó en pantalla del **servidor encontrado** (mostrando su IP interna en la red pruebas y el puerto). Con ello quedó demostrado que los tres servidores recibieron el broadcast y que el cliente seleccionó al primero que respondió para enviar el mensaje “HOLA”.  
   (Captura recomendada: salida del cliente con el texto “Servidor encontrado: (IP, puerto) → SERVIDOR ACTIVO”.)
7. **Verificación final (opcional)**
   * Identificación de la IP de cada servidor dentro de la red pruebas.
   * Consulta de los registros de cada servidor para observar la recepción del mensaje de descubrimiento y, en el elegido, la petición “HOLA”.  
     (Capturas recomendadas: inspección de contenedores con sus IPs internas y, si se desea, extractos de logs.)

**Evidencias incluidas**

* Listado de contenedores con los tres servidores activos en pruebas.
* Inspección de la red mostrando la subred **172.18.0.0/16**.
* Salida del cliente indicando el servidor activo (IP interna y puerto).
* (Opcional) Inspecciones/logs de servidores donde se aprecia la recepción del broadcast.

**Conclusión**

Se ha puesto en marcha un escenario UDP con **descubrimiento por broadcast** en una red Docker aislada. El cliente difundió a la dirección de broadcast **172.18.255.255**, recibió respuesta de los servidores y estableció comunicación con el primero que contestó. El ejercicio confirma el funcionamiento del broadcast en una red Docker, la resolución por nombres dentro de la misma red y el uso de contenedores para aislar servicios y pruebas.

Principio del formulario

Pensando

## Parte 2:

### EJ-1: Servidor y cliente TCP con envío secuencial

### EJ-2: CP con recvall y sendall para transmisión fiable

### EXPERIMENTO: TCP: efecto del tamaño de mensaje y sincronización

### EJ-4

### EJ-5

### EJ-6

### EJ-7