SISTEMAS DISTRIBUIDOS MEMORIA

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

Índice

[Práctica 1 3](#_Toc210571516)

[Parte 1: 3](#_Toc210571517)

[EJ-1: Servidor y Cliente UDP con parámetros por defecto 3](#_Toc210571518)

[EJ-2: UDP con pérdida simulada y numeración de mensajes 4](#_Toc210571519)

[EJ-3: UDP con confirmación y detección de pérdida 6](#_Toc210571520)

[EJ-4: Cliente UDP con reintentos y timeout progresivo 8](#_Toc210571521)

[EJ-5: UDP con control de duplicados y reintentos inteligentes 10](#_Toc210571522)

[EJ-6: Descubrimiento de servidores UDP por broadcast 12](#_Toc210571523)

[EJ-7: Descubrimiento UDP por broadcast en Docker 13](#_Toc210571524)

[Parte 2: 17](#_Toc210571525)

[EJ-1: Servidor y cliente TCP con envío secuencial 17](#_Toc210571526)

[**Objetivo** 17](#_Toc210571527)

[**Entorno** 17](#_Toc210571528)

[**Procedimiento realizado** 17](#_Toc210571529)

[**Resultados observados** 18](#_Toc210571530)

[**Evidencias (capturas propuestas)** 18](#_Toc210571531)

[**Conclusión** 18](#_Toc210571532)

[EJ-2: CP con recvall y sendall para transmisión fiable 18](#_Toc210571533)

[**Objetivo** 18](#_Toc210571534)

[**Entorno** 19](#_Toc210571535)

[**Procedimiento realizado** 19](#_Toc210571536)

[**Resultados observados** 19](#_Toc210571537)

[**Evidencias (capturas propuestas)** 19](#_Toc210571538)

[**Conclusión** 19](#_Toc210571539)

[EXPERIMENTO: TCP: efecto del tamaño de mensaje y sincronización 20](#_Toc210571540)

[EJ-4 21](#_Toc210571541)

[EJ-5 21](#_Toc210571542)

[EJ-6 21](#_Toc210571543)

[EJ-7 21](#_Toc210571544)

# Práctica 1

## Parte 1:

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

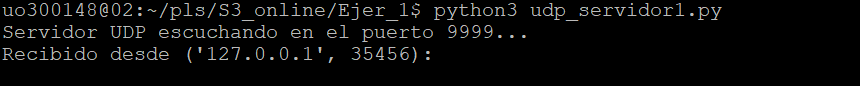
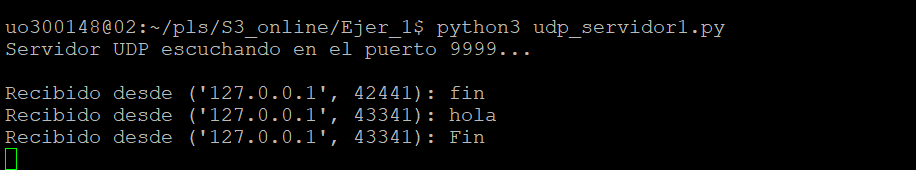


foto ejecutando la prueba en la ventana de la maquina virtual





**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.

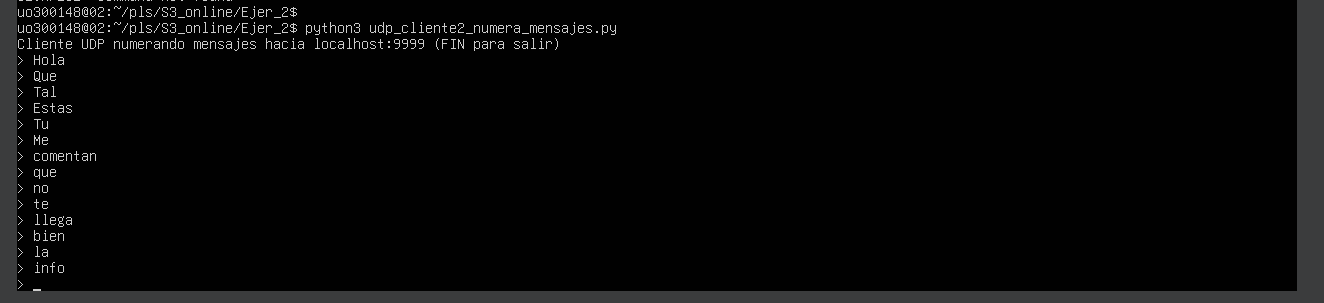
**Observación:** dado que UDP es un protocolo no orientado a conexión, no se garantiza la entrega, pero en las pruebas realizadas en entorno local todos los datagramas fueron recibidos correctamente.

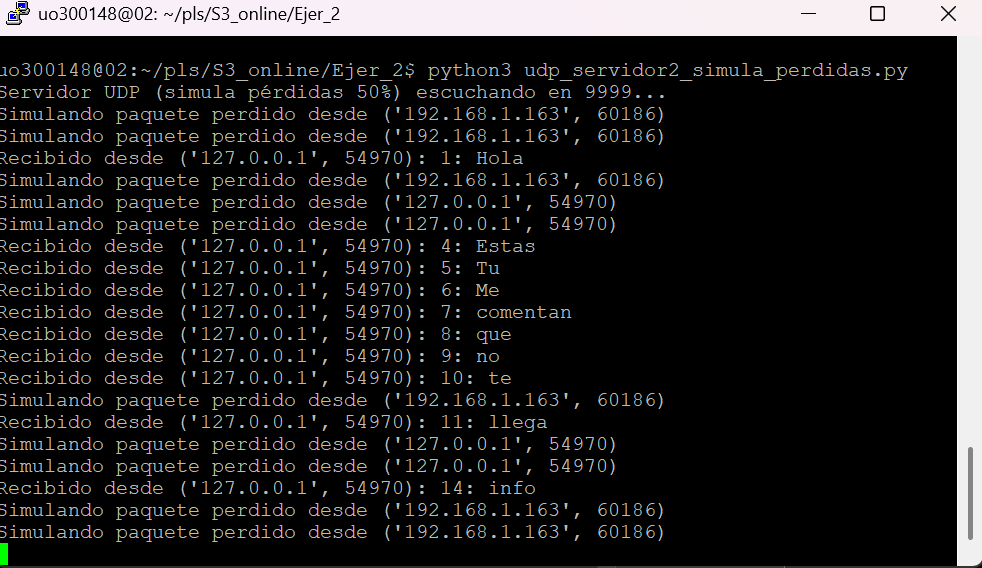
**4. Reflexiones y conclusiones**

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.

### 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**

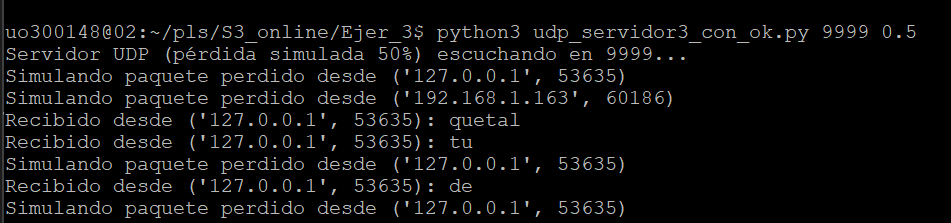
* El servidor mostró correctamente tanto los mensajes recibidos como los que se marcaron como “paquete perdido”.
* En la secuencia de números enviados por el cliente se observaron **saltos** (ejemplo: se reciben 1, 3, 5), lo cual indica que los datagramas 2 y 4 se perdieron en la simulación.
* Al interrumpir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

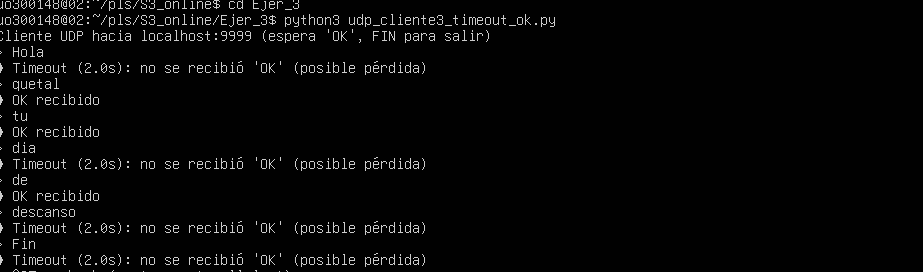
**4. Reflexiones y conclusiones**

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**

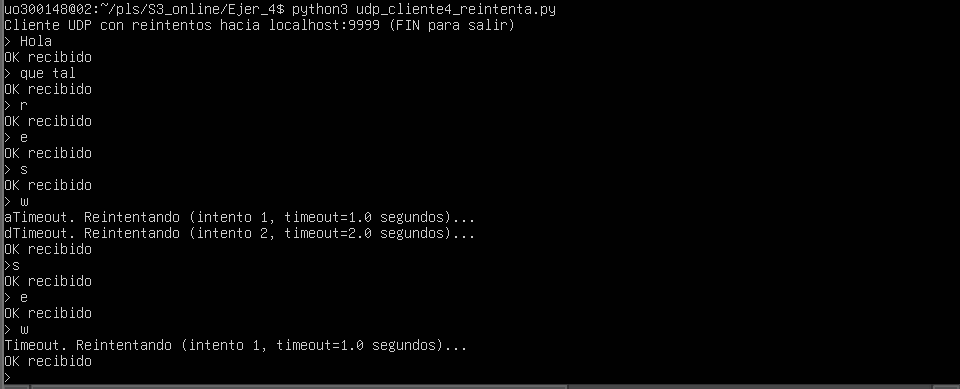
* Cuando no hay pérdida, el cliente recibe el "OK" del servidor en menos de 2 segundos.
* Cuando el servidor “pierde” un datagrama, el cliente muestra un timeout y sabe que el envío no fue confirmado.
* También se observó que, si el "OK" se pierde, el cliente se comporta igual que si se hubiese perdido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

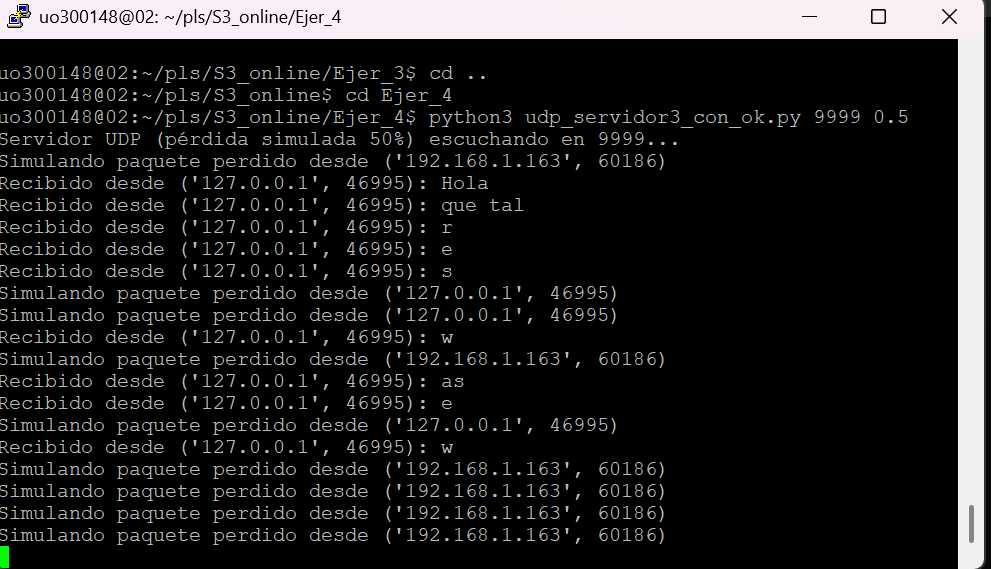
**4. Reflexiones y conclusiones**

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**

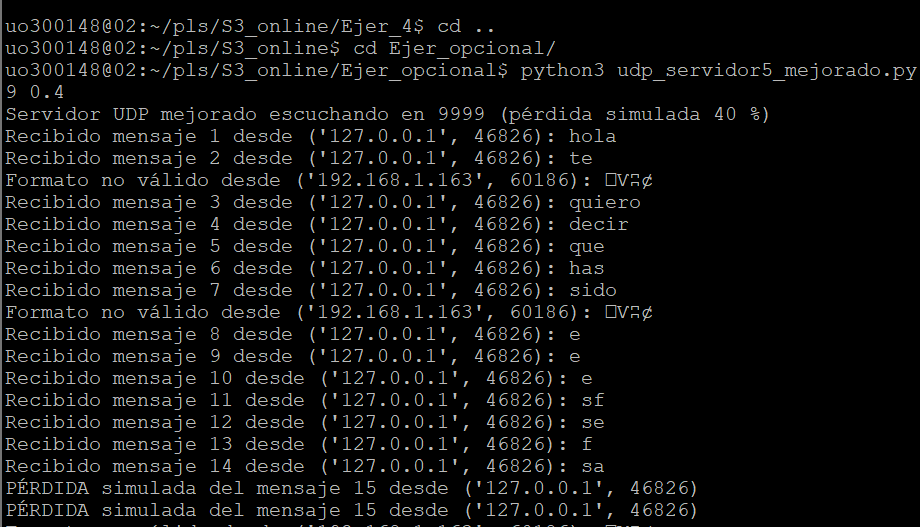
* Cuando el servidor recibe y responde, el cliente muestra el mensaje:
* OK recibido
* Cuando no se recibe el "OK", el cliente repite el envío, aumentando progresivamente el tiempo de espera.
* Tras varios intentos, si el tiempo de espera supera los 2 segundos, el cliente detiene los reintentos e informa:
* 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

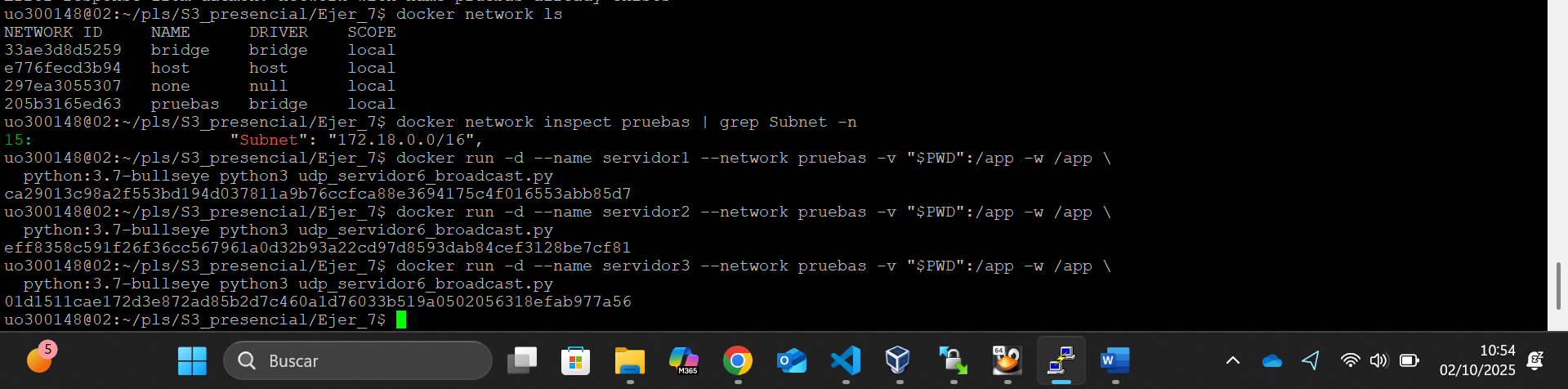
**Aplicaciones Prácticas**

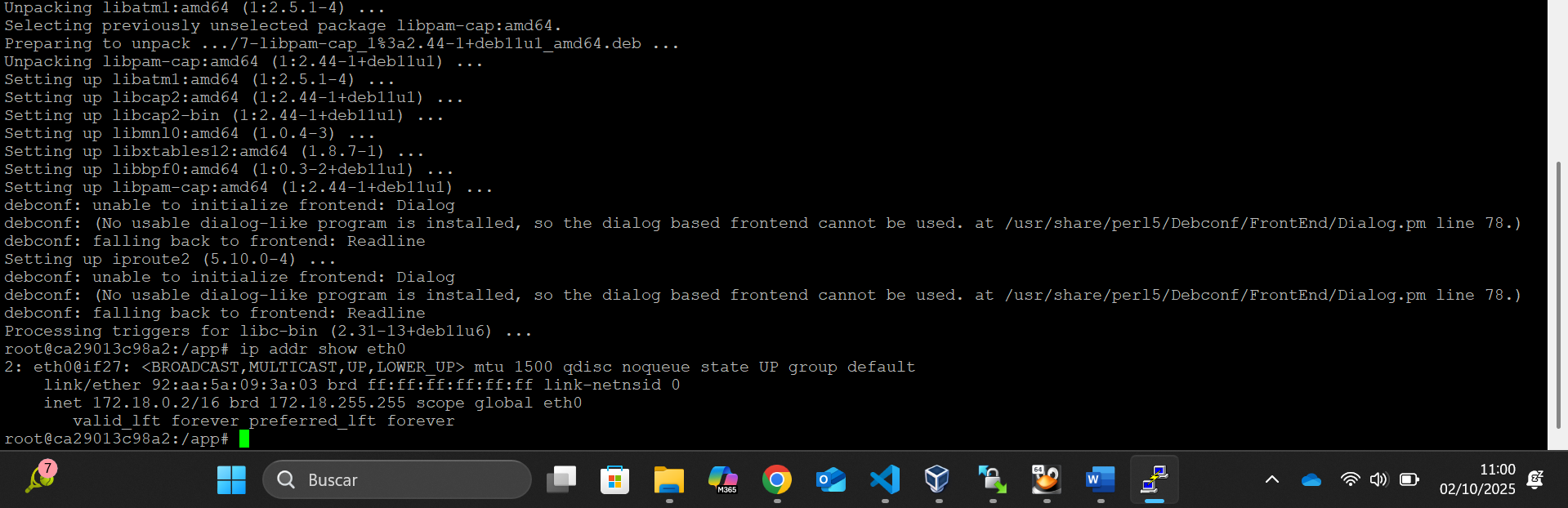
* Sistemas de monitorización distribuida
* Juegos en red local
* Servicios de impresión y recursos compartidos
* Entornos de cluster y computación distribuida

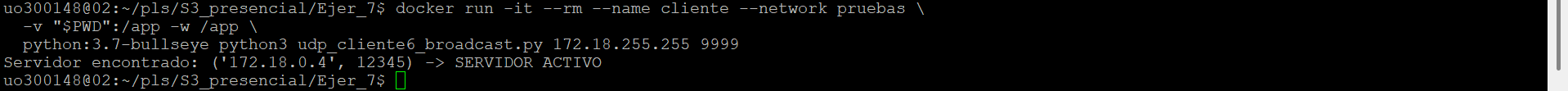
**Conclusión**

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.

**Comandos utilizados:**

# 1. LIMPIAR CONTENEDORES PREVIOS

docker stop $(docker ps -aq) 2>/dev/null

docker rm $(docker ps -aq) 2>/dev/null

docker network prune -f

docker system prune -f

# 2. CREAR RED DOCKER

docker network create pruebas

# 3. LANZAR 3 SERVIDORES

docker run -d --name servidor1 --network pruebas -v $(pwd):/app python:3.7 python /app/udp\_servidor6\_broadcast.py

docker run -d --name servidor2 --network pruebas -v $(pwd):/app python:3.7 python /app/udp\_servidor6\_broadcast.py

docker run -d --name servidor3 --network pruebas -v $(pwd):/app python:3.7 python /app/udp\_servidor6\_broadcast.py

# 4. VERIFICAR QUE ESTÁN EJECUTÁNDOSE

docker ps

# 5. AVERIGUAR DIRECCIÓN BROADCAST

docker run -it --network pruebas --rm python:3.7 bash

# Dentro del contenedor ejecutar:

apt-get update

apt-get install iproute2

ip addr show eth0

# Buscar línea: inet 172.18.0.5/16 brd 172.18.255.255

# Broadcast = 172.18.255.255

exit

# 6. LANZAR CLIENTE BROADCAST

docker run -it --network pruebas -v $(pwd):/app python:3.7 python /app/udp\_cliente6\_broadcast.py 172.18.255.255

# RESULTADO OBTENIDO:

# Buscando servidores...

# Servidor encontrado: ('172.18.0.4', 12345)

# Servidor encontrado: ('172.18.0.3', 12345)

# Servidor encontrado: ('172.18.0.2', 12345)

# Usando primer servidor: ('172.18.0.4', 12345)

# DIRECCIONES IP OBSERVADAS:

# - Cliente: 172.18.0.5

# - Servidor1: 172.18.0.2

# - Servidor2: 172.18.0.3

# - Servidor3: 172.18.0.4

# - Broadcast: 172.18.255.255

# COMANDOS PARA LIMPIEZA FINAL:

# docker stop servidor1 servidor2 servidor3

# docker rm servidor1 servidor2 servidor3

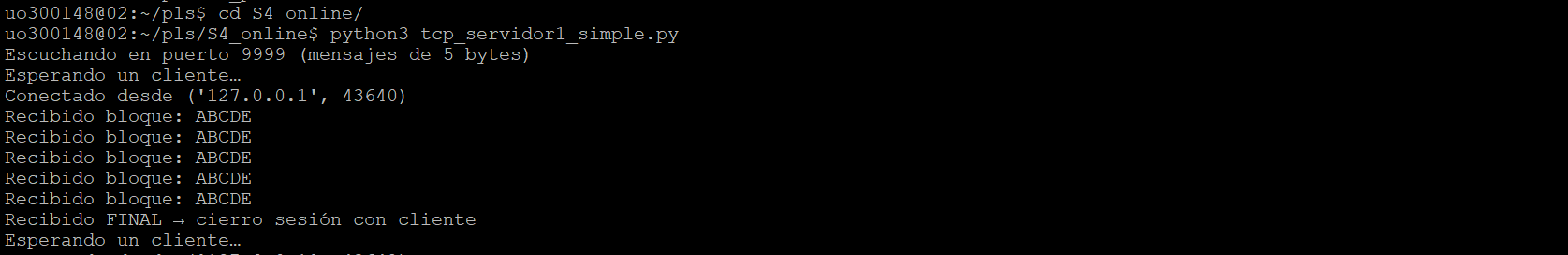
# docker network rm pruebas

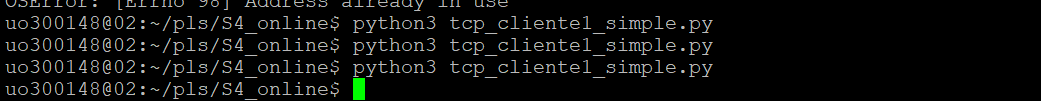
EOF

Principio del formulario

## Parte 2:

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





**Objetivo**

Implementar y probar un servidor TCP que atiende a un cliente a la vez y recibe mensajes de 5 bytes, finalizando la sesión cuando llega la palabra “FINAL”. Desarrollar un cliente que envía cinco bloques “ABCDE” (5 bytes) y después “FINAL”.

**Entorno**

VM Linux de prácticas.

Archivos: tcp\_servidor1\_simple.py (servidor) y tcp\_cliente1\_simple.py (cliente).

Pruebas locales con localhost.

**Procedimiento realizado**

Creación de los dos scripts en el PC (VS Code) y transferencia a la VM (WinSCP).

Ejecución del servidor en la VM, escuchando en un puerto TCP (por defecto 9999).

Ejecución del cliente en otra terminal de la VM, conectando a localhost y al mismo puerto.

El cliente envía cinco mensajes de 5 bytes (“ABCDE”) y, a continuación, “FINAL” para cerrar de forma ordenada.

El servidor, tras recibir “FINAL”, cierra la conexión con ese cliente y vuelve a quedar a la espera de nuevas conexiones.

**Resultados observados**

En la consola del servidor aparece “Conectado desde (127.0.0.1, puerto\_efímero)”.

Se muestran cinco líneas con “Recibido bloque: ABCDE”.

A continuación se muestra “Recibido FINAL → cierro sesión con cliente”.

El servidor vuelve a mostrar “Esperando un cliente…”, quedando listo para otra conexión.

Al relanzar el cliente, el patrón se repite con el mismo comportamiento.

**Evidencias (capturas propuestas)**

Servidor en estado “Escuchando / Esperando un cliente”.

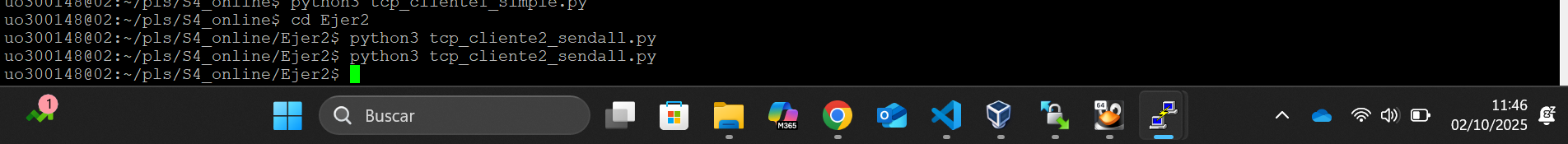
Secuencia de recepción en el servidor: 5× “Recibido bloque: ABCDE” y “Recibido FINAL → cierro sesión con cliente”.

Mensaje final de “Esperando un cliente…” tras cerrar la sesión.

**Conclusión**

Queda verificado el esquema “tamaño prefijado” en TCP: el servidor procesa bloques fijos de 5 bytes y usa “FINAL” como señal de terminación, manteniéndose operativo para aceptar nuevos clientes después de cada sesión.

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





**Objetivo**

Mejorar el ejercicio anterior para hacer el intercambio robusto en TCP:

En el cliente, garantizar que cada mensaje se envía completo usando sendall().

En el servidor, leer exactamente N bytes (N=5) con una función recvall(sock, n) que acumula lecturas hasta completar el tamaño esperado.

**Entorno**

VM Linux de prácticas, mismos directorios del tema.

Archivos realizados: tcp\_servidor2\_recvall.py (servidor con recvall) y tcp\_cliente2\_sendall.py (cliente con sendall).

Pruebas locales cliente-servidor en la misma máquina (localhost).

**Procedimiento realizado**

Servidor: se incorporó la función recvall(sock, 5) para leer el tamaño prefijado. El servidor acepta una conexión, entra en un bucle y, por cada iteración, usa recvall para obtener justo 5 bytes; si recibe “FINAL”, cierra la sesión y vuelve a esperar otro cliente.

Cliente: se sustituyó el envío manual por sendall() para garantizar el envío íntegro de cada bloque. El cliente envía cinco mensajes “ABCDE” (5 bytes) y, a continuación, “FINAL”.

Ejecución: servidor y cliente se lanzan en sesiones separadas; se observa en la consola del servidor la recepción ordenada de los bloques y el cierre tras “FINAL”.

**Resultados observados**

El servidor muestra “Conectado desde …”.

Aparecen cinco líneas con “Recibido bloque: ABCDE”.

Después se muestra “Recibido FINAL → cerrar” y el servidor queda de nuevo en “Esperando un cliente…”.

La comunicación mantiene la sincronía incluso si el flujo TCP fracciona envíos o recepciones, porque sendall() asegura el envío completo y recvall() acumula hasta N bytes.

**Evidencias (capturas propuestas)**

Consola del servidor con: conexión entrante, cinco recepciones de “ABCDE”, recepción de “FINAL” y vuelta al estado de espera.

Consola del cliente finalizando tras completar los envíos.

**Conclusión**

El uso combinado de sendall() (emisor) y recvall() (receptor) hace el protocolo de tamaño prefijado robusto frente al fraccionamiento natural del stream TCP. Se confirma que el servidor recibe exactamente 5 bytes por mensaje, detecta “FINAL” correctamente y queda listo para atender nuevas conexiones.

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

Pantalla de computadora

El contenido generado por IA puede ser incorrecto.

**3.1. Diseño del Experimento**

Se diseñó un escenario controlado donde:

* **Cliente**: Envía sistemáticamente 4 bytes ("ABCD") en lugar de los 5 bytes esperados
* **Servidor**: Utiliza recvall(5) que bloquea hasta recibir exactamente 5 bytes
* **Objetivo**: Analizar cómo TCP maneja el desfase entre tamaños de envío y recepción

**3.2. Comportamiento Observado**

**Fase inicial (primeros envíos):**

* Cliente envía "ABCD" (4 bytes) × 5 veces = 20 bytes totales
* Servidor en recvall(5) se bloquea tras el primer envío, esperando 1 byte adicional
* El buffer TCP acumula: "ABCD" + "ABCD" = "ABCDABCD" (8 bytes)

**Procesamiento distorsionado:**

* Primer retorno de recvall(): "ABCDA" (mezcla del final del primer mensaje con inicio del segundo)
* Segundo retorno: "BCDA" + "B" = "BCDAB" (continúa la mezcla de bytes entre mensajes)
* Los límites originales de los mensajes se pierden completamente

**Detección corrupta de FINAL:**

* El mensaje "FINAL" (5 bytes) llega pero no en los límites esperados
* Ejemplo real observado: "AFINA" + "L" + "CD" (mezcla con bytes residuales)
* El servidor detecta "FINAL" de forma fragmentada o embebida en otros datos

**3.3. Análisis Técnico**

**Causa raíz:**

* recvall(5) insiste en leer 5 bytes exactos, ignorando los límites lógicos de mensaje
* TCP provee un flujo continuo de bytes sin delimitadores inherentes
* La acumulación en buffer rompe la correspondencia 1:1 entre envíos y recepciones

**Impacto en la aplicación:**

* **Falso positivo**: Detección de "FINAL" en posición incorrecta
* **Pérdida de sincronización**: Protocolo de comunicación se corrompe
* **Bloqueo potencial**: Si el cliente envía menos datos de los esperados

**3.4. Conclusión Experimental**

El experimento demostró que recvall()**es un arma de doble filo**: mientras garantiza recepción completa, su uso rígido con tamaños fijos puede distorsionar completamente la semántica de la comunicación cuando existe desalineación entre las expectativas del cliente y servidor. La lección fundamental es que cualquier protocolo sobre TCP debe incluir mecanismos explícitos de delimitación o indicación de longitud.

### EJ-3

**Objetivo**  
Implementar un servicio "oche" básico que invierte cadenas usando TCP con tamaño fijo de mensaje (80 bytes) y delimitadores \r\n.

**Entorno**  
VM Linux de prácticas, mismos directorios del tema.  
Archivos realizados: tcp\_servidor3\_oche\_simplista.py y tcp\_cliente3\_oche.py.  
Pruebas locales cliente-servidor en la misma máquina (localhost).

**Procedimiento realizado**

* Servidor: Crea socket TCP, acepta conexiones y en bucle recibe hasta 80 bytes por mensaje. Detecta \r\n como fin de mensaje, invierte el contenido y responde con el texto invertido más \r\n.
* Cliente: Se conecta al servidor y envía varias líneas de texto terminadas en \r\n, mostrando por pantalla cada respuesta recibida.
* Ejecución: Servidor y cliente se lanzan en sesiones separadas; se observa en consola el intercambio de mensajes y sus versiones invertidas.

**Resultados observados**

* El servidor muestra "Conectado desde..." al aceptar cliente.
* Por cada mensaje recibido, muestra "Recibido: 'TEXTO\r\n'" y "Enviando: 'OTXET\r\n'".
* El cliente muestra cada mensaje enviado y la respuesta invertida recibida.
* La comunicación funciona correctamente para mensajes individuales enviados con pausas.

**Evidencias (capturas propuestas)**

* Consola del servidor mostrando recepción y envío de mensajes invertidos.
* Consola del cliente mostrando el ciclo completo de envío-recepción.

**Conclusión**  
El enfoque simplista funciona para mensajes individuales, pero es vulnerable a la concatenación de mensajes en el buffer TCP cuando se envían rápidamente, ya que recv(80) puede recibir múltiples mensajes juntos.

### EJ-4

**Objetivo**  
Mejorar el servicio OCHE usando readline() para delimitar mensajes automáticamente mediante \r\n, manejando correctamente mensajes concatenados.

**Entorno**  
VM Linux de prácticas, mismos directorios del tema.  
Archivos realizados: tcp\_servidor4\_oche\_readline.py y tcp\_cliente4\_oche\_readline.py.  
Pruebas locales cliente-servidor en la misma máquina.

**Procedimiento realizado**

* Servidor: Usa makefile() para convertir el socket en objeto archivo con encoding UTF-8 y delimitador \r\n. Implementa readline() para leer líneas completas automáticamente.
* Cliente: Similar al anterior pero también usa makefile() y readline() para recibir respuestas.
* Se añade time.sleep(1) en servidor para simular condiciones de red donde mensajes se concatenan.
* Ejecución: Servidor y cliente se lanzan en sesiones separadas; se envían múltiples mensajes rápidamente.

**Resultados observados**

* El servidor recibe cada mensaje individualmente aunque lleguen concatenados en buffer.
* Se muestra "Recibido: 'TEXTO\r\n'" por cada línea completa.
* Las respuestas se envían correctamente como texto invertido.
* El cliente recibe cada respuesta individualmente.

**Evidencias (capturas propuestas)**

* Consola del servidor mostrando recepción ordenada de múltiples mensajes.
* Consola del cliente mostrando respuestas correctas a pesar de envíos rápidos.

**Conclusión**  
El uso de readline() resuelve el problema de mensajes concatenados, haciendo el protocolo robusto frente a la bufferización TCP. La comunicación mantiene la integridad de mensajes incluso con envíos rápidos y múltiples.

### EJ-5

**Objetivo**  
Implementar un servicio OCHE que use longitud prefijada para delimitar mensajes, eliminando la dependencia de caracteres especiales como delimitadores.

**Entorno**  
VM Linux de prácticas, mismos directorios del tema.  
Archivos realizados: tcp\_servidor5\_oche\_len.py y tcp\_cliente5\_oche\_len.py.  
Pruebas locales cliente-servidor en la misma máquina.

**Procedimiento realizado**

* Servidor: Implementa función recibe\_longitud() que lee hasta \n para obtener la longitud, luego recibe exactamente esa cantidad de bytes. Responde con el texto invertido, enviando primero su longitud.
* Cliente: Para cada mensaje, envía primero la longitud en bytes seguida de \n, luego el mensaje. Usa recibe\_longitud() para leer la longitud de la respuesta.
* Formato: [longitud]\n[mensaje] para ambos lados.
* Ejecución: Servidor y cliente se lanzan en sesiones separadas; se intercambian varios mensajes.

**Resultados observados**

* El servidor muestra "Enviando longitud: '4\n'" y "Enviando mensaje: 'ALOH'" para mensaje "HOLA".
* El cliente muestra el mismo formato para envíos y recepciones.
* La comunicación es precisa, sin dependencia de contenido del mensaje.
* Maneja correctamente mensajes de cualquier longitud y contenido.

**Evidencias (capturas propuestas)**

* Consola del servidor mostrando el protocolo de longitud+mensaje.
* Consola del cliente mostrando el ciclo completo con longitud explícita.

**Conclusión**  
El método de longitud prefijada es el más robusto, ya que no depende de caracteres especiales y maneja cualquier contenido binario o textual. Aunque más complejo, garantiza delimitación precisa en todos los casos.

### EJ-6

**Objetivo**  
Desplegar el servicio OCHE en contenedores Docker con mapeo de puertos, permitiendo acceso desde fuera del contenedor.

**Entorno**  
VM Linux de prácticas con Docker instalado.  
Archivos utilizados: tcp\_servidor4\_oche\_readline.py y tcp\_cliente4\_oche\_readline.py.

**Procedimiento realizado**

* Crear red Docker pruebas para comunicación entre contenedores.
* Lanzar servidor en contenedor con mapeo de puertos -p 9090:9999.
* Conectar cliente desde otro contenedor en la misma red.
* Conectar cliente desde host local usando puerto mapeado.
* Probar conexión desde máquina externa usando IP del servidor.

**Resultados observados**

* Servidor en contenedor acepta conexiones tanto de otros contenedores como del exterior.
* Cliente en contenedor puede conectar usando nombre del servicio (servidor\_oche).
* Cliente externo puede conectar usando IP y puerto mapeado (9090).
* El servicio funciona idénticamente dentro y fuera de contenedores.

**Evidencias (capturas propuestas)**

* Salida de docker ps mostrando mapeo de puertos.
* Consola de servidor en contenedor mostrando conexiones entrantes.
* Consola de cliente externo mostrando comunicación exitosa.

**Conclusión**  
Docker permite desplegar servicios TCP de forma aislada pero accesible, usando redes internas para comunicación entre microservicios y mapeo de puertos para acceso externo. El servicio OCHE funciona transparentemente en este entorno.