SISTEMAS DISTRIBUIDOS MEMORIA

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

Índice

[Práctica 1 3](#_Toc210928298)

[Parte 1: 3](#_Toc210928299)

[EJ-1: Servidor y Cliente UDP con parámetros por defecto 3](#_Toc210928300)

[EJ-2: UDP con pérdida simulada y numeración de mensajes 4](#_Toc210928301)

[EJ-3: UDP con confirmación y detección de pérdida 6](#_Toc210928302)

[EJ-4: Cliente UDP con reintentos y timeout progresivo 7](#_Toc210928303)

[EJ-5: UDP con control de duplicados y reintentos inteligentes 9](#_Toc210928304)

[EJ-6: Descubrimiento de servidores UDP por broadcast 11](#_Toc210928305)

[EJ-7: Descubrimiento UDP por broadcast en Docker 12](#_Toc210928306)

[Parte 2: 16](#_Toc210928307)

[EJ-1: Servidor y cliente TCP con envío secuencial 16](#_Toc210928308)

[EJ-2: CP con recvall y sendall para transmisión fiable 17](#_Toc210928309)

[EXPERIMENTO: TCP: efecto del tamaño de mensaje y sincronización 18](#_Toc210928310)

[EJ-3: Servidor TCP con puerto configurable y respuesta invertida 19](#_Toc210928311)

[EJ-4: Comunicación TCP con lectura por líneas y sincronización temporal 21](#_Toc210928312)

[EJ-5: Servidor y cliente TCP con protocolo basado en longitud de mensaje 22](#_Toc210928313)

[EXPERIMENTO: Conexión remota entre cliente y servidor TCP en red local 23](#_Toc210928314)

# Práctica 1

## Parte 1:

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

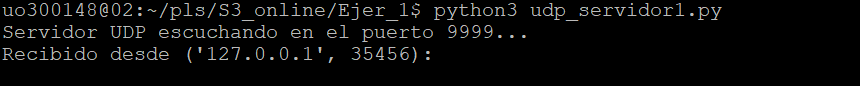
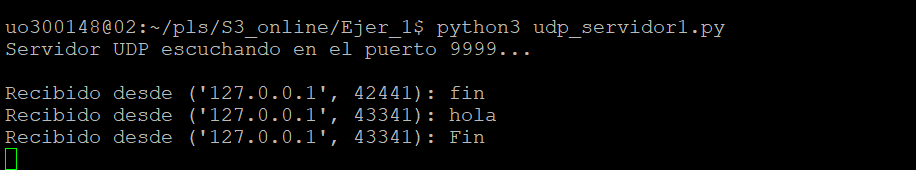


foto ejecutando la prueba en la ventana de la maquina virtual





**1. Introducción**

En esta práctica empezamos a trabajar con sockets UDP usando Python. El objetivo era entender cómo funciona este protocolo, creando un servidor y un cliente básicos que pudieran comunicarse entre sí. También usamos netcat para hacer pruebas rápidas y comprobar que los datagramas se enviaban y recibían correctamente.

**2. Desarrollo del ejercicio**

**Servidor UDP** Se creó un script en Python llamado udp\_servidor1.py que:

Escucha en el puerto 9999 por defecto.

Se queda esperando mensajes en un bucle infinito.

Muestra por pantalla el contenido del mensaje y la IP del cliente que lo envió. El código está disponible en el repositorio del grupo.

**3. Pruebas realizadas**

Se probó el servidor con netcat para verificar que recibía mensajes.

Se hizo la prueba cliente-servidor usando dos terminales en la misma máquina virtual.

Se comprobó que el servidor puede recibir varios mensajes seguidos sin problema. Nota: Aunque UDP no garantiza la entrega de datos, en nuestras pruebas locales todos los mensajes llegaron correctamente.

**4. Conclusiones**

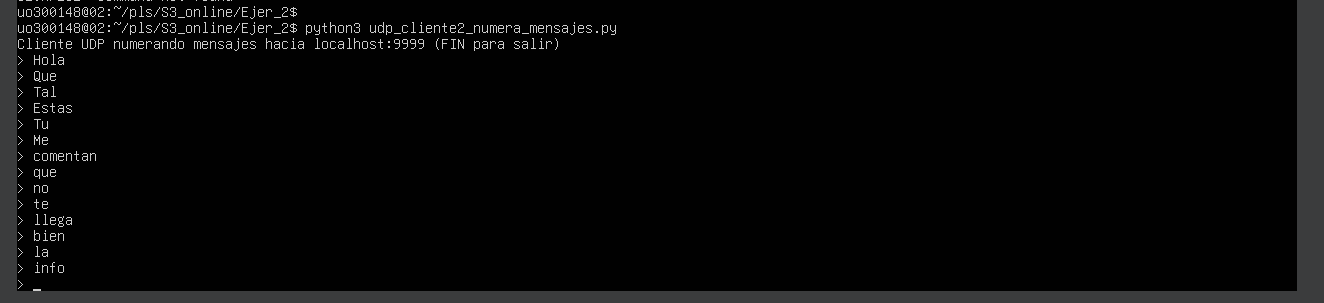
Gracias a este ejercicio:

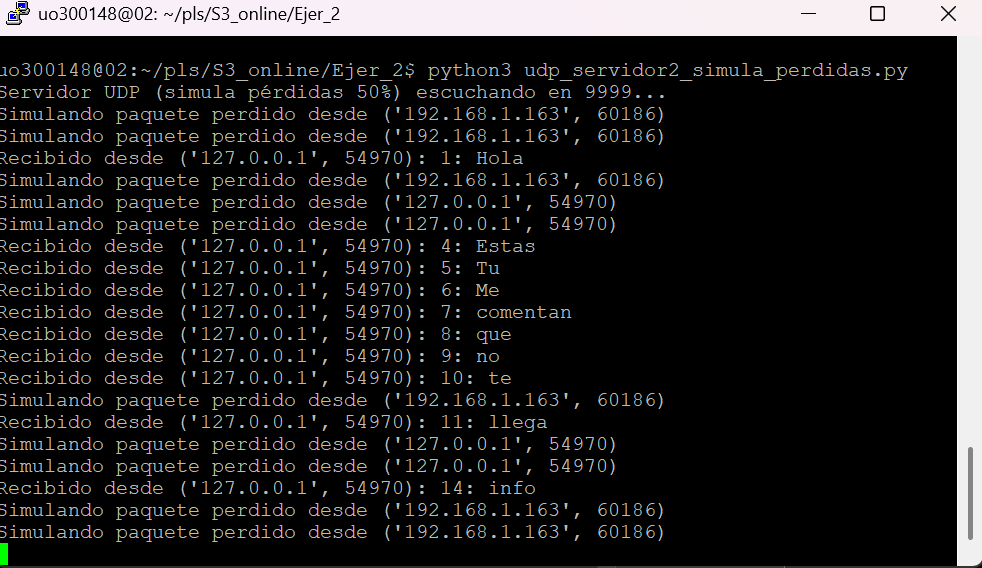
Entendimos mejor las diferencias entre TCP y UDP, sobre todo que UDP no necesita establecer conexión.

Aprendimos a crear sockets y enviar datagramas en Python.

Nos familiarizamos con netcat como herramienta útil para hacer pruebas de red.

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





**1. Introducción**

En este ejercicio se modificó el programa anterior para simular el comportamiento poco fiable de UDP. El objetivo era crear un servidor que perdiera mensajes de forma aleatoria y un cliente que numerara cada mensaje para detectar fácilmente cuándo se pierde alguno.

**2. Desarrollo del ejercicio**

**Servidor UDP con pérdidas simuladas** Se creó el script udp\_servidor2\_simula\_perdidas.py a partir del servidor del EJ-1.

Al recibir un datagrama, el servidor decide aleatoriamente (50% por defecto) si lo procesa o lo “pierde”.

Si se pierde, muestra el mensaje: Simulando paquete perdido desde (IP, puerto).

Si se recibe, muestra el contenido y la IP del cliente.

Se puede configurar la probabilidad de pérdida al ejecutar el programa.

Al cerrar con Ctrl+C, muestra estadísticas de cuántos paquetes se perdieron realmente.

**Cliente UDP con numeración de mensajes** Se creó el script udp\_cliente2\_numera\_mensajes.py, que añade numeración a los mensajes:

Cada datagrama empieza con un número, en formato "n: mensaje".

Esto permite ver fácilmente si se pierde alguno (por ejemplo, si se reciben 1, 3, 5).

El programa termina cuando se escribe FIN.

**3. Resultados observados**

El servidor mostró correctamente los mensajes recibidos y los que simuló como perdidos.

Se observaron saltos en la numeración, lo que indica que algunos datagramas no llegaron.

Al finalizar, el resumen de pérdidas coincidía bastante con la probabilidad configurada.

**4. Conclusiones**

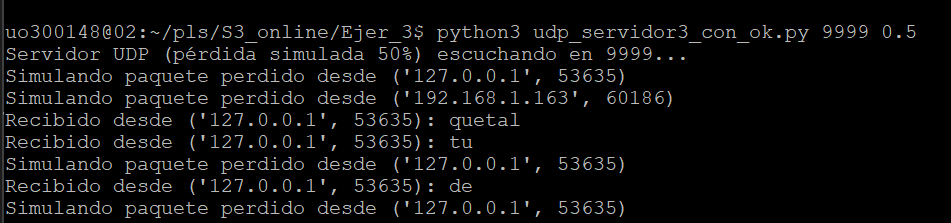
Este ejercicio ayudó a entender:

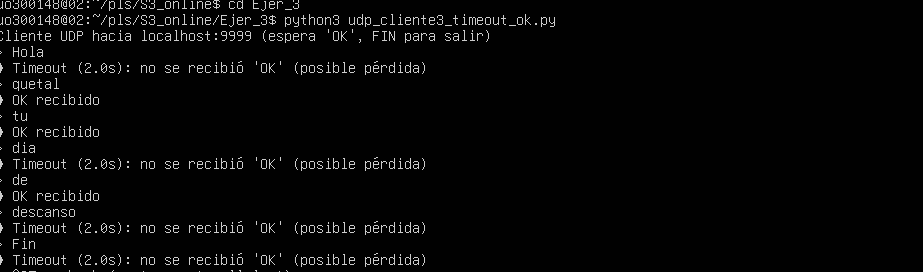
Cómo simular fallos de red para probar aplicaciones.

Que UDP no garantiza la entrega de datos.

La utilidad de numerar mensajes para detectar pérdidas, como hacen otros protocolos más avanzados.

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





**1. Introducción**

En este ejercicio se mejoró la comunicación entre cliente y servidor usando UDP para poder detectar si algún mensaje se pierde. La idea principal fue que el servidor enviara una confirmación ("OK") por cada datagrama recibido, y que el cliente usara un tiempo de espera para saber si su mensaje llegó o no.

**2. Desarrollo del ejercicio**

**Servidor UDP con confirmación** Se creó el script udp\_servidor3\_con\_ok.py, partiendo del servidor anterior:

Sigue simulando pérdidas con una probabilidad (50% por defecto).

Si se pierde un datagrama, se muestra el mensaje Simulando paquete perdido y no se responde.

Si se recibe correctamente, se muestra el contenido y se responde al cliente con "OK".

**Cliente UDP con espera de "OK"** Se creó el script udp\_cliente3\_timeout\_ok.py:

Envía mensajes al servidor y espera la respuesta "OK" usando recvfrom().

Se configuró un timeout de 2 segundos.

Si llega el "OK", se muestra ✓ OK recibido.

Si no llega nada en ese tiempo, se muestra ⏱ Timeout: no se recibió 'OK'.

**3. Resultados observados**

Cuando no hay pérdida, el cliente recibe el "OK" sin problema.

Si el servidor pierde el datagrama, el cliente detecta el fallo por el timeout.

También se vio que si el "OK" se pierde, el cliente lo interpreta igual que si se hubiera perdido el mensaje original, lo cual es realista en redes no fiables.

**4. Conclusiones**

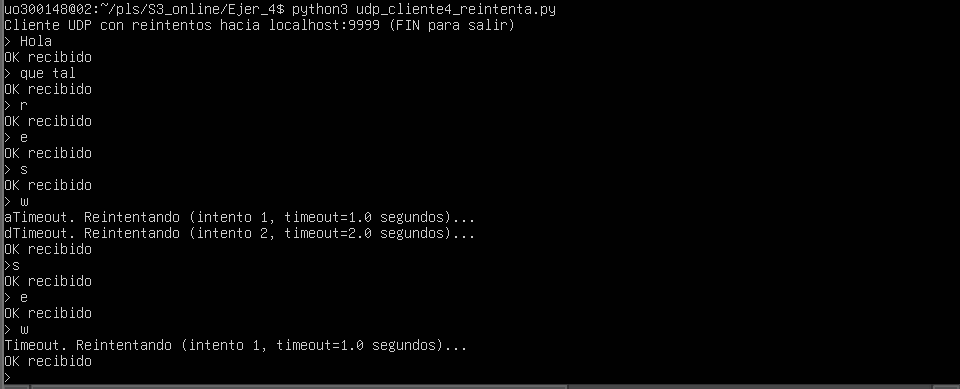
Este ejercicio sirvió para entender:

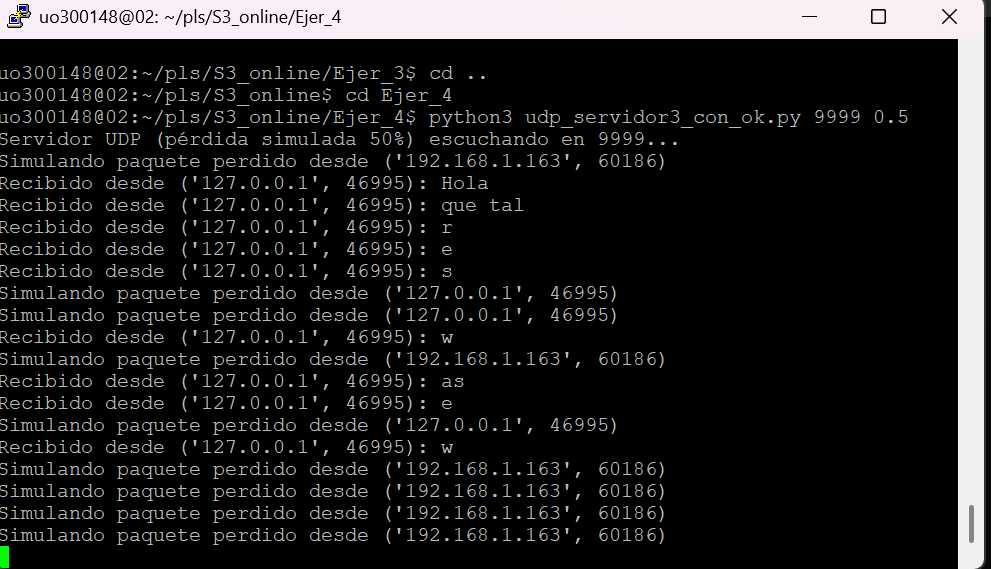
Que UDP no garantiza la entrega de datos.

Cómo implementar confirmaciones y timeouts para mejorar la fiabilidad.

Que este tipo de lógica es la base de cómo funcionan protocolos más avanzados como TCP.

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





**1. Introducción**

En esta práctica se mejoró el cliente UDP para que no solo detecte si un mensaje se pierde, sino que también lo reenvíe si no recibe confirmación. El objetivo era acercarse al funcionamiento de protocolos más fiables como TCP, usando reintentos y aumentando el tiempo de espera entre ellos.

**2. Desarrollo del ejercicio**

**Servidor empleado** Se utilizó el mismo servidor del EJ-3 (udp\_servidor3\_con\_ok.py), que responde con "OK" cuando recibe un datagrama correctamente.

**Cliente con reintentos** Se creó el script udp\_cliente4\_reintenta.py, basado en el cliente anterior:

Envía un mensaje y espera la respuesta "OK".

Si no recibe nada en el tiempo establecido, reenvía el mismo mensaje.

El timeout empieza en 0.5 segundos y se duplica en cada intento (0.5 → 1.0 → 2.0).

Si el tiempo supera los 2 segundos, el cliente asume que el servidor puede estar caído y muestra un aviso.

**3. Resultados observados**

Si el servidor responde, el cliente muestra OK recibido.

Si no hay respuesta, el cliente reintenta el envío aumentando el tiempo de espera.

Tras varios intentos, si no llega el "OK", el cliente detiene el proceso y muestra: Puede que el servidor esté caído. Inténtelo más tarde.

Así, el cliente evita quedarse bloqueado y puede actuar ante fallos.

**4. Conclusiones**

Este ejercicio ayudó a entender:

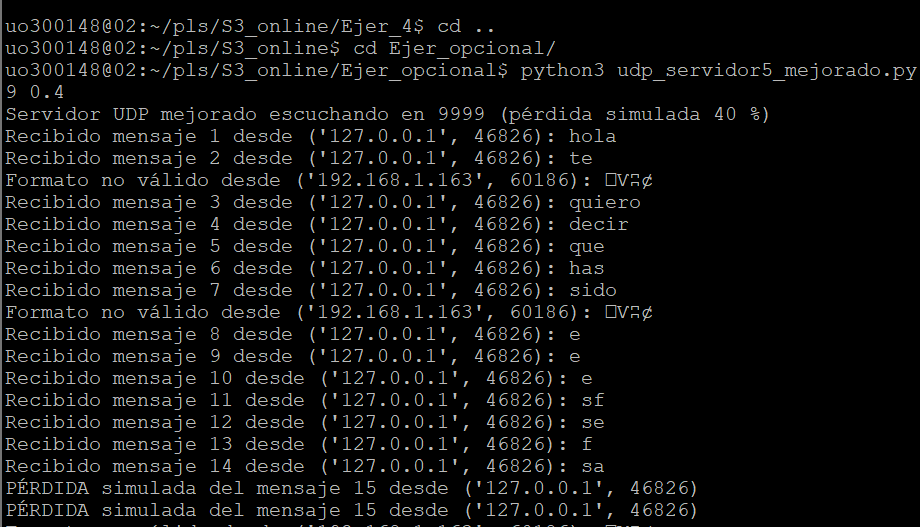
Cómo implementar reintentos y control de tiempo en UDP.

Que el cliente puede detectar fallos y reaccionar sin depender de una conexión estable.

Que este tipo de lógica se usa en protocolos confiables como TCP, donde hay confirmaciones, reenvíos y gestión de errores.

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





**1. Introducción**

Este ejercicio opcional se centró en mejorar la fiabilidad del protocolo UDP que veníamos desarrollando. El principal problema detectado era que el cliente no podía saber con certeza si la confirmación "OK" correspondía al mensaje enviado, lo que podía generar confusiones en caso de reintentos o llegada desordenada. Para solucionarlo, se añadieron identificadores únicos a cada mensaje y se implementó un control de duplicados en el servidor.

**2. Desarrollo del ejercicio**

**Mejoras en el cliente (**udp\_cliente5\_mejorado.py**)**

Cada mensaje se envía con un identificador aleatorio en formato ID|mensaje.

El cliente espera una confirmación específica del tipo OK <ID>.

Si no recibe respuesta, reintenta el envío del mismo mensaje, duplicando el tiempo de espera en cada intento (0.2 → 0.4 → 0.8 → 1.6...).

Si el timeout supera los 2 segundos, el cliente aborta el proceso y muestra un mensaje de error.

Solo avanza al siguiente mensaje si recibe la confirmación correcta.

**Mejoras en el servidor (**udp\_servidor5\_mejorado.py**)**

El servidor sigue simulando pérdidas con una probabilidad configurable.

Al recibir un datagrama, extrae el identificador y el contenido.

Si el mensaje ya fue procesado (duplicado), no lo ejecuta de nuevo, pero sí reenvía la confirmación correspondiente.

Si es nuevo, lo procesa y guarda el ID para evitar duplicados futuros.

En ambos casos, responde con OK <ID> al cliente.

**3. Resultados observados**

El cliente solo acepta confirmaciones que coincidan con el ID del mensaje enviado.

En caso de pérdida, el cliente reenvía el mensaje hasta recibir la confirmación o agotar los intentos.

El servidor detecta duplicados y evita procesarlos dos veces, lo que mejora la eficiencia y evita errores.

Se logró una comunicación más robusta, incluso en condiciones simuladas de pérdida.

**4. Conclusiones**

Este ejercicio permitió implementar un sistema de comunicación más fiable sobre UDP:

Los identificadores únicos permiten asociar cada mensaje con su confirmación.

El control de duplicados evita que el servidor repita acciones innecesarias.

El cliente gestiona los reintentos de forma inteligente, con timeout progresivo. Estas mejoras acercan el comportamiento del protocolo al de TCP, manteniendo la simplicidad y flexibilidad de UDP.

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

**Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.**

**1. Introducción**

En este ejercicio se implementó un sistema de descubrimiento automático de servicios en red local utilizando UDP broadcast. El objetivo era permitir que un cliente detecte servidores disponibles sin necesidad de conocer sus direcciones IP, lo que resulta útil en entornos dinámicos o distribuidos.

**2. Desarrollo del ejercicio**

**Servidor de Broadcast**

Escucha en un puerto configurable (por defecto 12345).

Responde con "HOLA" a cualquier mensaje que contenga "SERVICIO".

Permite que varios servidores funcionen simultáneamente en la misma red.

**Cliente de Broadcast**

Envía el mensaje "SERVICIO" a la dirección de broadcast 255.255.255.255.

Recoge todas las respuestas recibidas en un tiempo máximo de 3 segundos.

Selecciona automáticamente el primer servidor que responde.

Envía una confirmación al servidor elegido con el mensaje "HOLA cliente".

**Protocolo de comunicación**

Cliente → Broadcast: "SERVICIO"

Servidores → Cliente: "HOLA"

Cliente → Servidor elegido: "HOLA cliente"

**3. Resultados observados**

Se logró descubrir múltiples servidores en la red local de forma automática.

La comunicación funcionó correctamente entre la máquina virtual y el sistema anfitrión.

Los tiempos de respuesta fueron inferiores a 1 segundo.

El cliente manejó correctamente los casos en los que no había servidores disponibles.

**4. Aplicaciones prácticas**

Este tipo de descubrimiento puede aplicarse en:

Sistemas de monitorización distribuida.

Juegos multijugador en red local.

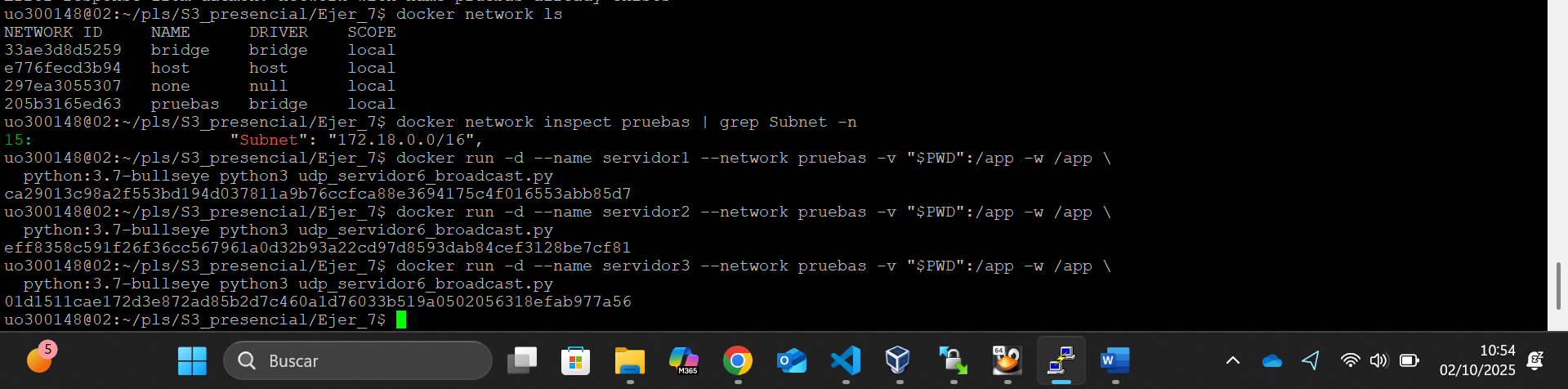
Servicios compartidos como impresión o archivos.

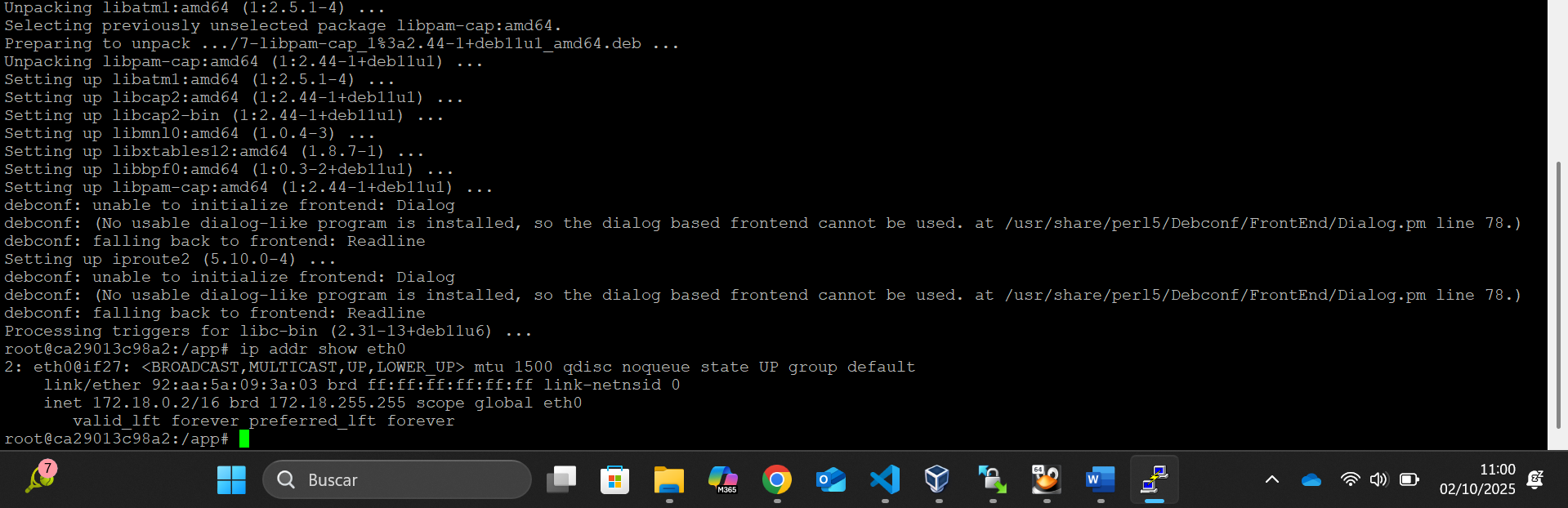
Clústeres y entornos de computación distribuida.

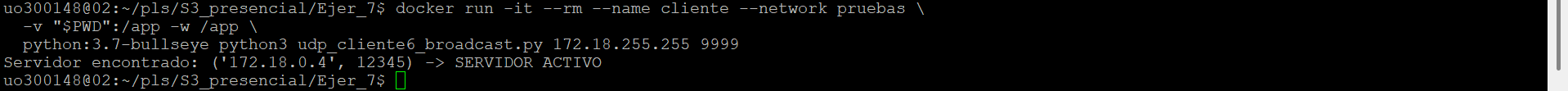
**5. Conclusiones**

El ejercicio demostró que el uso de UDP broadcast es una solución efectiva y sencilla para descubrir servicios en red local. Esta técnica permite escalar sistemas distribuidos sin necesidad de configuraciones manuales, facilitando la conexión entre clientes y servidores de forma dinámica.

### 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


**1. Introducción**

Este ejercicio consistió en desplegar varios servidores UDP dentro de una red Docker aislada y permitir que un cliente los descubriera automáticamente mediante broadcast. El objetivo era comprobar que el mecanismo de descubrimiento funciona correctamente en un entorno de contenedores, sin necesidad de conocer las IPs de los servidores.

**2. Entorno y preparación**

Se usó una carpeta de trabajo con los scripts del cliente y servidor.

La imagen base fue Python sobre Debian “bullseye”, con herramientas de red disponibles.

Se creó (o reutilizó) una red Docker llamada pruebas para aislar los contenedores y permitir resolución de nombres.

**3. Procedimiento realizado**

**1. Red de trabajo** Se utilizó la red pruebas para conectar todos los contenedores y simular una red local.

**2. Lanzamiento de servidores** Se iniciaron tres contenedores (servidor1, servidor2, servidor3) ejecutando el script del servidor en primer plano.

**3. Comprobación inicial** Se verificó que los tres servidores estuvieran activos y conectados a la red pruebas.

**4. Cálculo de broadcast** Se inspeccionó la red para obtener la subred 172.18.0.0/16, lo que permitió calcular la dirección de broadcast: 172.18.255.255.

**5. Ejecución del cliente** Se lanzó un contenedor con el script del cliente, que envió el mensaje "SERVICIO" a la IP de broadcast. El cliente recogió las respuestas y seleccionó el primer servidor que respondió.

**6. Resultado de la detección** El cliente mostró en pantalla las IPs y puertos de los servidores que respondieron, y confirmó cuál fue el primero en contestar.

**7. Verificación final (opcional)** Se inspeccionaron las IPs internas de cada servidor y se revisaron los logs para confirmar la recepción del mensaje de descubrimiento y la respuesta "HOLA".

**4. Evidencias incluidas**

Listado de contenedores activos.

Inspección de la red pruebas mostrando la subred.

Salida del cliente con los servidores detectados.

(Opcional) Logs de los servidores mostrando la recepción del broadcast.

**5. Conclusiones**

Este ejercicio demostró que el descubrimiento por broadcast funciona correctamente en una red Docker aislada. El cliente pudo detectar múltiples servidores sin conocer sus IPs, seleccionó uno automáticamente y estableció comunicación. Además, se confirmó que Docker permite la resolución de nombres y el uso de broadcast dentro de una red personalizada, lo que facilita el despliegue de sistemas distribuidos para pruebas o producción.

**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:

apt-get update

apt-get install iproute2

ip addr show eth0

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

# 7. LIMPIEZA FINAL

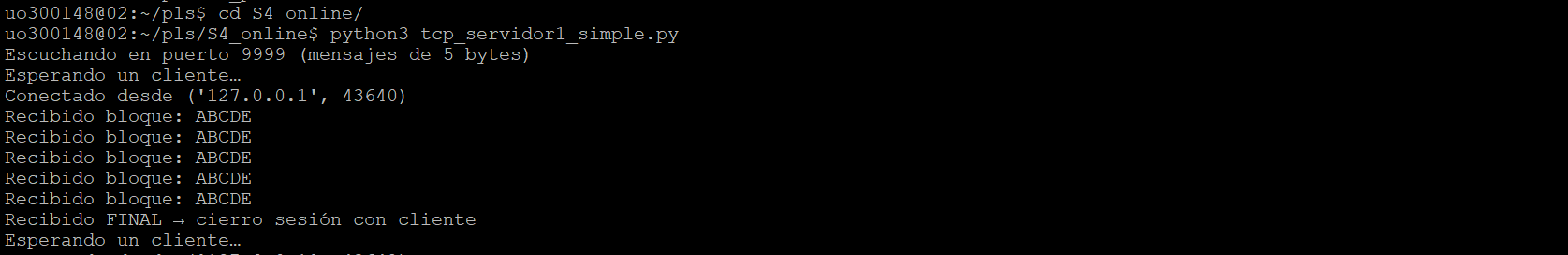
docker stop servidor1 servidor2 servidor3

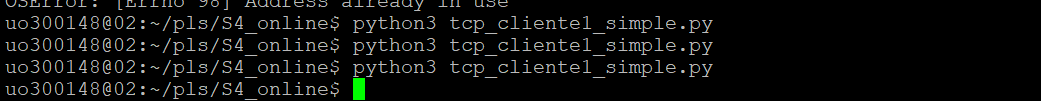
docker rm servidor1 servidor2 servidor3

docker network rm pruebasPrincipio del formulario

## Parte 2:

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





**1. Introducción**

Este ejercicio consistió en implementar un servidor TCP que atiende a un cliente por sesión y recibe bloques de 5 bytes. El cliente envía cinco bloques "ABCDE" y luego el mensaje "FINAL" para cerrar la conexión. El objetivo era entender cómo funciona la comunicación secuencial en TCP y cómo gestionar el cierre ordenado de sesiones.

**2. Desarrollo del ejercicio**

**Entorno de trabajo**

Máquina virtual Linux.

Scripts: tcp\_servidor1\_simple.py y tcp\_cliente1\_simple.py.

Pruebas realizadas en localhost.

**Procedimiento**

Se crearon los scripts en VS Code y se transfirieron a la VM usando WinSCP.

El servidor se ejecutó en la VM, escuchando en el puerto TCP 9999.

El cliente se lanzó desde otra terminal, conectando a localhost.

El cliente envió cinco bloques "ABCDE" y luego "FINAL".

Al recibir "FINAL", el servidor cerró la sesión y volvió a esperar otro cliente.

**3. Resultados observados**

El servidor mostró: Conectado desde (127.0.0.1, puerto\_efímero).

Se imprimieron cinco líneas con Recibido bloque: ABCDE.

Luego se mostró Recibido FINAL → cierro sesión con cliente.

El servidor volvió a quedar en estado Esperando un cliente….

Al relanzar el cliente, el comportamiento se repitió correctamente.

**4. Evidencias sugeridas**

Captura del servidor en estado Escuchando / Esperando un cliente.

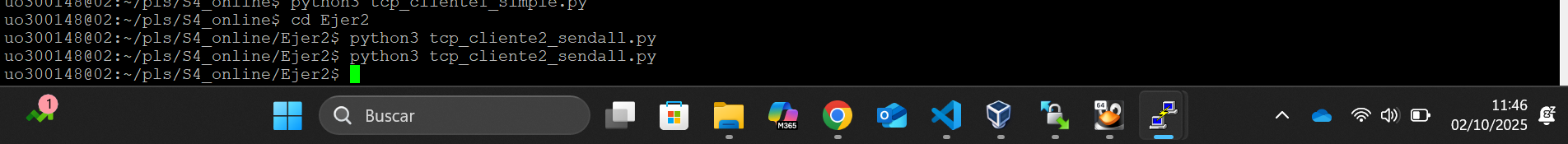
Secuencia de recepción: 5× "ABCDE" + "FINAL".

Mensaje final indicando que el servidor sigue activo.

**5. Conclusiones**

Este ejercicio confirmó el funcionamiento del esquema de bloques fijos en TCP. El servidor procesa mensajes de 5 bytes y reconoce "FINAL" como señal de cierre. La sesión se gestiona de forma ordenada y el servidor queda disponible para nuevas conexiones, cumpliendo con el modelo de atención secuencial.

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





**1. Introducción**

Este ejercicio mejora el anterior incorporando mecanismos para asegurar una transmisión robusta en TCP. El cliente utiliza sendall() para garantizar que cada bloque se envía completo, y el servidor emplea una función recvall() para leer exactamente 5 bytes por mensaje, incluso si el flujo TCP los entrega de forma fragmentada.

**2. Desarrollo del ejercicio**

**Entorno de trabajo**

Máquina virtual Linux.

Scripts: tcp\_servidor2\_recvall.py y tcp\_cliente2\_sendall.py.

Pruebas realizadas en localhost.

**Procedimiento**

El servidor incluye la función recvall(sock, 5) que acumula datos hasta recibir los 5 bytes esperados.

Acepta una conexión, entra en un bucle y procesa cada bloque recibido.

Si el bloque es "FINAL", cierra la sesión y vuelve a esperar otro cliente.

El cliente usa sendall() para enviar cinco bloques "ABCDE" y luego "FINAL", asegurando que cada envío se complete correctamente.

Ambos programas se ejecutan en terminales separadas.

**3. Resultados observados**

El servidor muestra: Conectado desde (IP, puerto).

Se imprimen cinco líneas con Recibido bloque: ABCDE.

Luego aparece Recibido FINAL → cerrar, y el servidor vuelve a Esperando un cliente….

La comunicación se mantiene sincronizada incluso si TCP fragmenta los datos, gracias a sendall() y recvall().

**4. Evidencias sugeridas**

Consola del servidor mostrando conexión, recepción de bloques, cierre por "FINAL" y retorno al estado de espera.

Consola del cliente mostrando el envío completo y cierre ordenado.

**5. Conclusiones**

El uso combinado de sendall() en el cliente y recvall() en el servidor garantiza una transmisión fiable en TCP, incluso ante fragmentación del flujo. Se confirma que el servidor recibe exactamente 5 bytes por mensaje, detecta "FINAL" correctamente y queda listo para atender nuevas conexiones sin errores.

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

Pantalla de computadora

El contenido generado por IA puede ser incorrecto.

**1. Introducción**

Este experimento se diseñó para analizar cómo afecta el desfase entre el tamaño de los mensajes enviados por el cliente y el tamaño esperado por el servidor en una comunicación TCP. El objetivo era observar el comportamiento del protocolo cuando el cliente envía bloques de 4 bytes y el servidor espera 5 bytes exactos mediante recvall().

**2. Diseño del experimento**

**Cliente**: Envía sistemáticamente bloques de 4 bytes ("ABCD").

**Servidor**: Utiliza recvall(5) para leer exactamente 5 bytes por mensaje.

**Objetivo**: Evaluar cómo TCP maneja la desalineación entre envío y recepción.

**3. Comportamiento observado**

**Fase inicial**

El cliente envía "ABCD" cinco veces (total: 20 bytes).

El servidor se bloquea tras el primer envío, esperando el quinto byte.

El buffer TCP acumula datos: "ABCDABCD".

**Procesamiento distorsionado**

Primer retorno de recvall(): "ABCDA" (mezcla de dos bloques).

Segundo retorno: "BCDAB" (continuación del desfase).

Los límites originales de los mensajes se pierden.

**Detección corrupta de** "FINAL"

El mensaje "FINAL" llega, pero no en los límites esperados.

Ejemplo observado: "AFINA", "L", "CD" (fragmentado y mezclado).

El servidor puede detectar "FINAL" de forma errónea o incompleta.

**4. Análisis técnico**

**Causa raíz**

recvall(5) fuerza la lectura de 5 bytes, ignorando la estructura lógica del mensaje.

TCP entrega un flujo continuo de bytes, sin delimitadores.

El buffer se acumula y rompe la correspondencia entre envíos y recepciones.

**Impacto en la aplicación**

Posibles falsos positivos al detectar "FINAL" fuera de contexto.

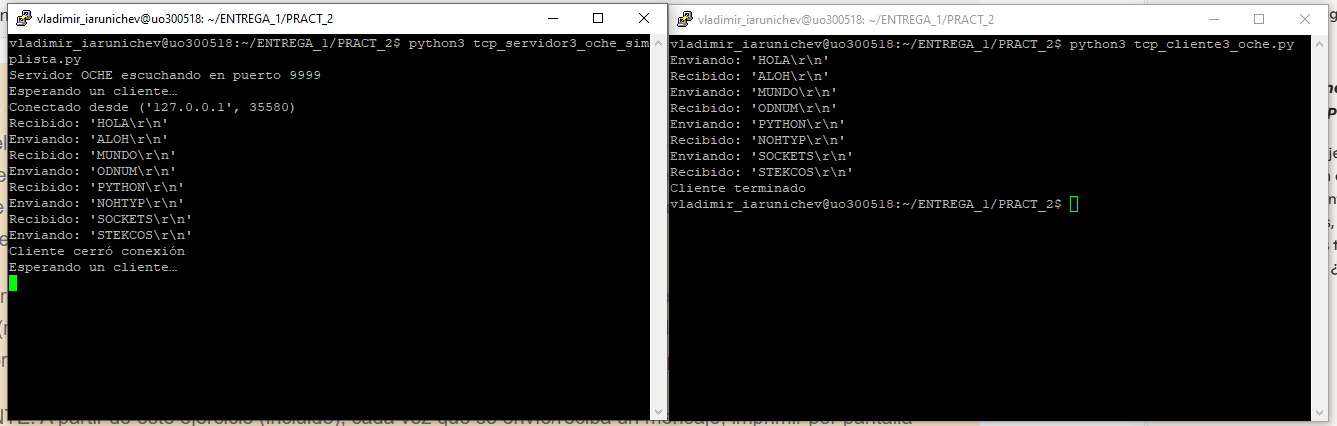
Pérdida de sincronización entre cliente y servidor.

Riesgo de bloqueo si el cliente envía menos datos de los esperados.

**5. Conclusión**

El experimento demuestra que el uso rígido de recvall() puede comprometer la semántica de la comunicación si no se respetan los tamaños acordados. TCP no delimita mensajes por sí solo, por lo que cualquier protocolo sobre TCP debe incluir mecanismos explícitos de delimitación o longitud para evitar errores de interpretación y mantener la sincronización.

### EJ-3: Servidor TCP con puerto configurable y respuesta invertida



**1. Introducción**

En este ejercicio se desarrolló un servicio TCP llamado “oche” que invierte cadenas de texto. El servidor recibe mensajes de hasta 80 bytes terminados en \r\n, los invierte y responde al cliente con el texto invertido seguido también de \r\n. El objetivo era practicar el manejo de delimitadores y el procesamiento de cadenas en un entorno TCP.

**2. Desarrollo del ejercicio**

**Entorno de trabajo**

Máquina virtual Linux.

Scripts: tcp\_servidor3\_oche\_simplista.py y tcp\_cliente3\_oche.py.

Pruebas realizadas en localhost.

**Procedimiento**

El servidor crea un socket TCP, acepta conexiones y entra en un bucle de recepción.

Por cada mensaje recibido (máximo 80 bytes), detecta el delimitador \r\n, invierte el contenido y responde con el texto invertido más \r\n.

El cliente se conecta al servidor y envía varias líneas de texto terminadas en \r\n, mostrando en pantalla cada respuesta recibida.

Ambos programas se ejecutan en sesiones separadas.

**3. Resultados observados**

El servidor muestra: Conectado desde (IP, puerto).

Por cada mensaje, imprime:

Recibido: 'TEXTO\r\n'

Enviando: 'OTXET\r\n'

El cliente muestra cada mensaje enviado y la respuesta invertida.

La comunicación funciona correctamente cuando los mensajes se envían con pausas.

**4. Evidencias sugeridas**

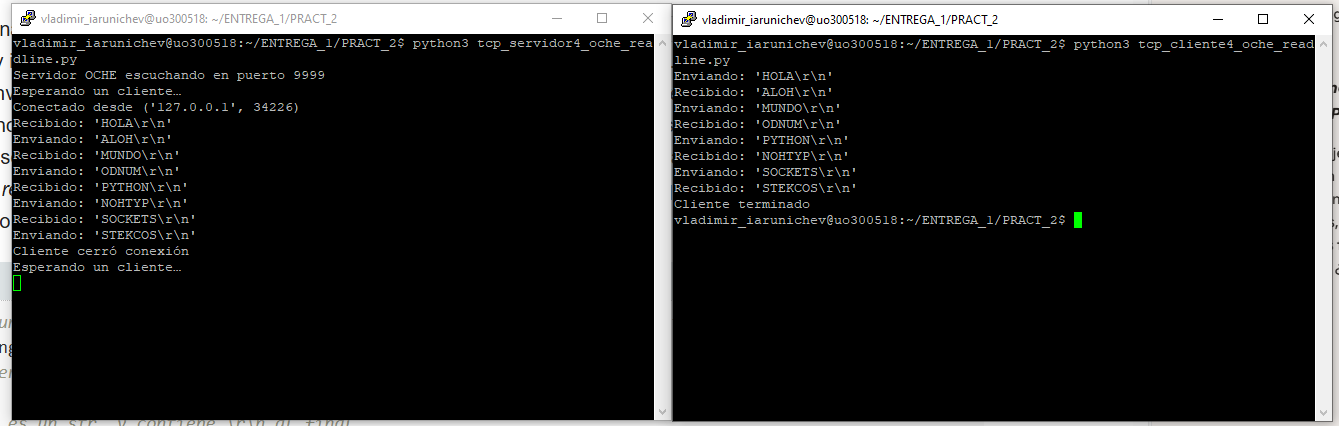
Consola del servidor mostrando la recepción y envío de mensajes invertidos.

Consola del cliente mostrando el ciclo completo de envío y recepción.

**5. Conclusiones**

El servicio funciona bien para mensajes individuales enviados con cierta pausa, pero presenta vulnerabilidades si se envían varios mensajes seguidos rápidamente. Esto se debe a que recv(80) puede recibir múltiples mensajes concatenados en el buffer TCP, lo que complica la detección de los delimitadores y puede afectar la lógica de inversión.

### EJ-4: Comunicación TCP con lectura por líneas y sincronización temporal



**1. Introducción**

Este ejercicio mejora el servicio OCHE implementado anteriormente, incorporando readline() para delimitar mensajes automáticamente mediante \r\n. El objetivo era resolver el problema de mensajes concatenados en el buffer TCP, asegurando que cada línea se procese de forma independiente, incluso cuando se envían rápidamente.

**2. Desarrollo del ejercicio**

**Entorno de trabajo**

Máquina virtual Linux.

Scripts: tcp\_servidor4\_oche\_readline.py y tcp\_cliente4\_oche\_readline.py.

Pruebas realizadas en localhost.

**Procedimiento**

El servidor convierte el socket en un objeto tipo archivo usando makefile(), con codificación UTF-8 y delimitador \r\n.

Utiliza readline() para leer líneas completas automáticamente.

Se añade time.sleep(1) para simular condiciones de red donde los mensajes pueden llegar concatenados.

El cliente también usa makefile() y readline() para enviar y recibir mensajes correctamente.

Se lanzan servidor y cliente en sesiones separadas, enviando múltiples mensajes seguidos.

**3. Resultados observados**

El servidor recibe cada mensaje de forma individual, incluso si llegan juntos en el buffer.

Por cada línea, muestra: Recibido: 'TEXTO\r\n'.

Las respuestas se envían correctamente como texto invertido.

El cliente recibe cada respuesta por separado, sin errores ni mezclas.

**4. Evidencias sugeridas**

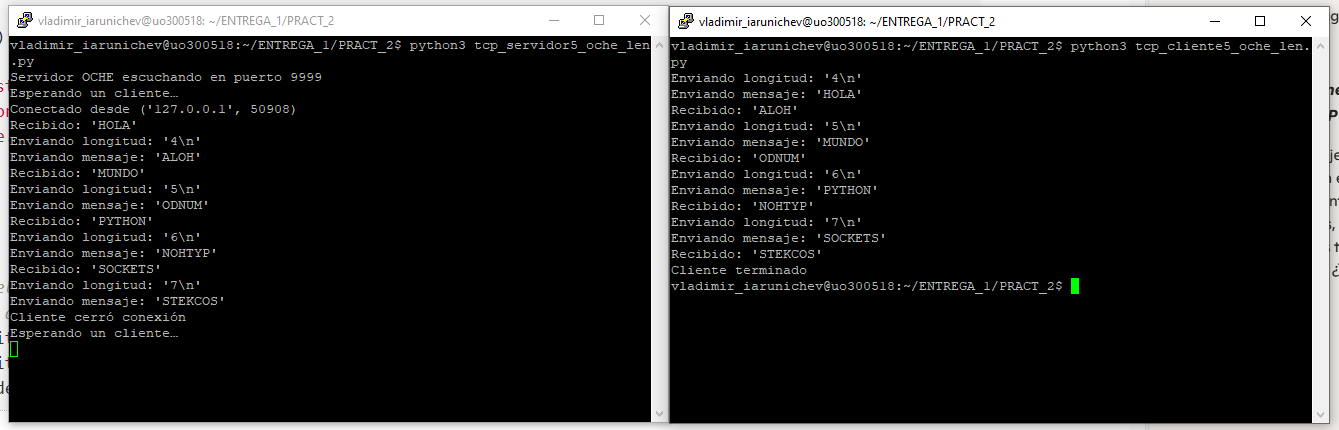
Consola del servidor mostrando la recepción ordenada de múltiples mensajes.

Consola del cliente mostrando respuestas correctas, incluso con envíos rápidos.

**5. Conclusiones**

El uso de readline() junto con makefile() permite manejar correctamente la delimitación de mensajes en TCP. Esta técnica hace que el protocolo sea más robusto frente a la bufferización natural del stream, manteniendo la integridad de cada mensaje incluso cuando se envían varios seguidos sin pausas.

### EJ-5: Servidor y cliente TCP con protocolo basado en longitud de mensaje



**1. Introducción**

Este ejercicio consistió en implementar una versión del servicio OCHE que utiliza un protocolo basado en longitud prefijada para delimitar los mensajes. El objetivo fue eliminar la dependencia de caracteres especiales como \r\n, permitiendo una comunicación más robusta y compatible con cualquier tipo de contenido.

**2. Desarrollo del ejercicio**

**Entorno de trabajo**

Máquina virtual Linux.

Scripts: tcp\_servidor5\_oche\_len.py y tcp\_cliente5\_oche\_len.py.

Pruebas realizadas en localhost.

**Procedimiento**

El servidor implementa la función recibe\_longitud() que primero lee hasta \n para obtener la longitud del mensaje, y luego lee exactamente esa cantidad de bytes.

Una vez recibido el mensaje, lo invierte y responde al cliente usando el mismo formato: primero la longitud del mensaje invertido seguida de \n, y luego el contenido.

El cliente, por su parte, envía cada mensaje indicando primero su longitud en bytes seguida de \n, y luego el texto.

Ambos programas se ejecutan en sesiones separadas y se intercambian varios mensajes.

**3. Resultados observados**

El servidor muestra:

Enviando longitud: '4\n'

Enviando mensaje: 'ALOH' (respuesta a "HOLA")

El cliente muestra el mismo formato en sus envíos y recepciones.

La comunicación es precisa y no depende del contenido del mensaje.

Se manejan correctamente mensajes de cualquier longitud, incluyendo aquellos que contienen caracteres especiales o binarios.

**4. Evidencias sugeridas**

Consola del servidor mostrando el protocolo de longitud + mensaje.

Consola del cliente mostrando el ciclo completo de envío y recepción con longitud explícita.

**5. Conclusiones**

El protocolo basado en longitud prefijada es el más robusto para delimitar mensajes en TCP. Al no depender de caracteres especiales, permite transmitir cualquier tipo de contenido de forma segura y precisa. Aunque requiere una lógica adicional para interpretar la longitud, garantiza una delimitación exacta y evita problemas de sincronización o fragmentación en el stream TCP.

### EXPERIMENTO: Conexión remota entre cliente y servidor TCP en red local

SOLO SE HACE EN EL AULA

**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.