

SISTEMAS DISTRIBUIDOS MEMORIA

Autores: Vladimir Iarunichev y Francisco Rebolo Cabo

Índice

Práctica 1.....	3
Parte 1:	3
EJ-1: Servidor y Cliente UDP con parámetros por defecto.....	3
EJ-2: UDP con pérdida simulada y numeración de mensajes	4
EJ-3: UDP con confirmación y detección de pérdida.....	6
EJ-4: Cliente UDP con reintentos y timeout progresivo	7
EJ-5: UDP con control de duplicados y reintentos inteligentes.....	9
EJ-6: Descubrimiento de servidores UDP por broadcast	11
EJ-7: Descubrimiento UDP por broadcast en Docker.....	12
Parte 2:	16
EJ-1: Servidor y cliente TCP con envío secuencial	16
EJ-2: CP con recvall y sendall para transmisión fiable	17
EXPERIMENTO: TCP: efecto del tamaño de mensaje y sincronización	18
EJ-3: Servidor TCP con puerto configurable y respuesta invertida.....	20
EJ-4: Comunicación TCP con lectura por líneas y sincronización temporal	21
EJ-5: Servidor y cliente TCP con protocolo basado en longitud de mensaje	22
EXPERIMENTO: Conexión remota entre cliente y servidor TCP en red local	24
Parte 3:	26
EJ-1: Servidor TCP para calcular números primos.....	26
EJ-2: Programación concurrente con select()	28
EJ-3: Programación paralela con procesos (fork)	29

Práctica 1

Parte 1:

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

```
uo300148@02:~/pls/S3_online/Ejer_1$ python3 udp_servidor1.py
Servidor UDP escuchando en el puerto 9999...
Recibido desde ('127.0.0.1', 35456):
```

```
uo300148@02:~/pls/S3_online$ cd Ejer_1/
uo300148@02:~/pls/S3_online/Ejer_1$ nc -u 127.0.0.1 9999
```

```
uo300148@02:~/pls/S3_online/Ejer_1$ python3 udp_cliente1.py
Cliente UDP enviando a localhost:9999 (escribe FIN para salir)
> fin
> FIN
uo300148@02:~/pls/S3_online/Ejer_1$ python3 udp_cliente1.py
Cliente UDP enviando a localhost:9999 (escribe FIN para salir)
> hola
> Fin
>
```

```
uo300148@02:~/pls/S3_online/Ejer_1$ python3 udp_servidor1.py
Servidor UDP escuchando en el puerto 9999...

Recibido desde ('127.0.0.1', 42441): fin
Recibido desde ('127.0.0.1', 43341): hola
Recibido desde ('127.0.0.1', 43341): Fin
```

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.

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

```
U0300140802:~/p1s/S3_online/Ejer_2$
U0300140802:~/p1s/S3_online/Ejer_2$ python3 udo_cliente2_numeram_mensajes.py
Cliente UDP numerando mensajes hacia localhost:9999 (FIN para salir)
> Hola
> Que
> Tal
> Estas
> Tu
> Me
> comentan
> que
> no
> te
> llega
> bien
> la
> info
>
> _
```

```
uo300148@02: ~/pls/S3_online/Ejer_2
python3 udp_servidor2_simula_perdidas.py
Servidor UDP (simula pérdidas 50%) escuchando en 9999...
Simulando paquete perdido desde ('192.168.1.163', 60186)
Simulando paquete perdido desde ('192.168.1.163', 60186)
Recibido desde ('127.0.0.1', 54970): 1: Hola
Simulando paquete perdido desde ('192.168.1.163', 60186)
Simulando paquete perdido desde ('127.0.0.1', 54970)
Simulando paquete perdido desde ('127.0.0.1', 54970)
Recibido desde ('127.0.0.1', 54970): 4: Estas
Recibido desde ('127.0.0.1', 54970): 5: Tu
Recibido desde ('127.0.0.1', 54970): 6: Me
Recibido desde ('127.0.0.1', 54970): 7: comentan
Recibido desde ('127.0.0.1', 54970): 8: que
Recibido desde ('127.0.0.1', 54970): 9: no
Recibido desde ('127.0.0.1', 54970): 10: te
Simulando paquete perdido desde ('192.168.1.163', 60186)
Recibido desde ('127.0.0.1', 54970): 11: llega
Simulando paquete perdido desde ('127.0.0.1', 54970)
Simulando paquete perdido desde ('127.0.0.1', 54970)
Recibido desde ('127.0.0.1', 54970): 14: info
Simulando paquete perdido desde ('192.168.1.163', 60186)
Simulando paquete perdido desde ('192.168.1.163', 60186)
```

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, muestras estadísticas de cuántos paquetes se perdieron realmente.

Cliente UDP con numeración de mensajes Se creó el script

`udp_cliente2_numerar_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

```
uo300148002:~/pls/S3_online/Ejer_3$ python3 udp_servidor3_con_ok.py 9999 0.5
Servidor UDP (pérdida simulada 50%) escuchando en 9999...
Simulando paquete perdido desde ('127.0.0.1', 53635)
Simulando paquete perdido desde ('192.168.1.163', 60186)
Recibido desde ('127.0.0.1', 53635): quetal
Recibido desde ('127.0.0.1', 53635): tu
Simulando paquete perdido desde ('127.0.0.1', 53635)
Recibido desde ('127.0.0.1', 53635): de
Simulando paquete perdido desde ('127.0.0.1', 53635)

uo300148002:~/pls/S3_online$ cd Ejer_3
uo300148002:~/pls/S3_online/Ejer_3$ python3 udp_cliente3_timeout_ok.py
Cliente UDP hacia localhost:9999 (espera 'OK', FIN para salir)
Hola
Timeout (2.0s): no se recibió 'OK' (posible pérdida)
quetal
OK recibido
tu
OK recibido
dia
Timeout (2.0s): no se recibió 'OK' (posible pérdida)
de
OK recibido
descanso
Timeout (2.0s): no se recibió 'OK' (posible pérdida)
Fin
Timeout (2.0s): no se recibió 'OK' (posible 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

```
lup000148@02:~/pls/S3_online/Ejer_4$ python3 udp_cliente4_reintenta.py
Cliente UDP con reintentos hacia localhost:9999 (FIN para salir)
> Hola
OK recibido
> que tal
OK recibido
> r
OK recibido
> e
OK recibido
> s
OK recibido
> w
aTimeout. Reintentando (intento 1, timeout=1.0 segundos)...
dTimeout. Reintentando (intento 2, timeout=2.0 segundos)...
OK recibido
> s
OK recibido
> e
OK recibido
> w
Timeout. Reintentando (intento 1, timeout=1.0 segundos)...
OK recibido
>
```

```
uo300148@02: ~/pls/S3_online/Ejer_4
uo300148@02:~/pls/S3_online/Ejer_3$ cd ..
uo300148@02:~/pls/S3_online$ cd Ejer_4
uo300148@02:~/pls/S3_online/Ejer_4$ python3 udp_servidor3_con_ok.py 9999 0.5
Servidor UDP (pérdida simulada 50%) escuchando en 9999...
Simulando paquete perdido desde ('192.168.1.163', 60186)
Recibido desde ('127.0.0.1', 46995): Hola
Recibido desde ('127.0.0.1', 46995): que tal
Recibido desde ('127.0.0.1', 46995): r
Recibido desde ('127.0.0.1', 46995): e
Recibido desde ('127.0.0.1', 46995): s
Simulando paquete perdido desde ('127.0.0.1', 46995)
Simulando paquete perdido desde ('127.0.0.1', 46995)
Recibido desde ('127.0.0.1', 46995): w
Simulando paquete perdido desde ('192.168.1.163', 60186)
Recibido desde ('127.0.0.1', 46995): as
Recibido desde ('127.0.0.1', 46995): e
Simulando paquete perdido desde ('127.0.0.1', 46995)
Recibido desde ('127.0.0.1', 46995): w
Simulando paquete perdido desde ('192.168.1.163', 60186)
Simulando paquete perdido desde ('192.168.1.163', 60186)
Simulando paquete perdido desde ('192.168.1.163', 60186)
Simulando paquete perdido desde ('192.168.1.163', 60186)
```

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 reintentará 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

```
uo300148002:~/pls/S3_online/Ejer_opcional$ python3 udp_cliente5_mejorado.py
Cliente UDP mejorado hacia localhost:9999 (escriba FIN para salir)
> hola
Confirmación recibida para el mensaje 1
> te
Confirmación recibida para el mensaje 2
> quiero
Confirmación recibida para el mensaje 3
> decir
Confirmación recibida para el mensaje 4
> que
Confirmación recibida para el mensaje 5
> has
Confirmación recibida para el mensaje 6
> sido
Confirmación recibida para el mensaje 7
> e
Confirmación recibida para el mensaje 8
> e
Confirmación recibida para el mensaje 9
> e
Confirmación recibida para el mensaje 10
> sf
Confirmación recibida para el mensaje 11
> se
Confirmación recibida para el mensaje 12
> f
Confirmación recibida para el mensaje 13
> sa
Confirmación recibida para el mensaje 14
> d
Timeout. Reintentando con timeout 1.0 segundos...
No se recibió confirmación. Puede que el servidor esté caído o haya pérdidas persistentes.
>
```

```

uo300148@02:~/pls/S3_online/Ejer_4$ cd ..
uo300148@02:~/pls/S3_online$ cd Ejer_opcional/
uo300148@02:~/pls/S3_online/Ejer_opcional$ python3 udp_servidor5_mejorado.py
9 0.4
Servidor UDP mejorado escuchando en 9999 (pérdida simulada 40 %)
Recibido mensaje 1 desde ('127.0.0.1', 46826): hola
Recibido mensaje 2 desde ('127.0.0.1', 46826): te
Formato no válido desde ('192.168.1.163', 60186): ☐V☐☐
Recibido mensaje 3 desde ('127.0.0.1', 46826): quiero
Recibido mensaje 4 desde ('127.0.0.1', 46826): decir
Recibido mensaje 5 desde ('127.0.0.1', 46826): que
Recibido mensaje 6 desde ('127.0.0.1', 46826): has
Recibido mensaje 7 desde ('127.0.0.1', 46826): sido
Formato no válido desde ('192.168.1.163', 60186): ☐V☐☐
Recibido mensaje 8 desde ('127.0.0.1', 46826): e
Recibido mensaje 9 desde ('127.0.0.1', 46826): e
Recibido mensaje 10 desde ('127.0.0.1', 46826): e
Recibido mensaje 11 desde ('127.0.0.1', 46826): sf
Recibido mensaje 12 desde ('127.0.0.1', 46826): se
Recibido mensaje 13 desde ('127.0.0.1', 46826): f
Recibido mensaje 14 desde ('127.0.0.1', 46826): sa
PÉRDIDA simulada del mensaje 15 desde ('127.0.0.1', 46826)
PÉRDIDA simulada del mensaje 15 desde ('127.0.0.1', 46826)

```

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

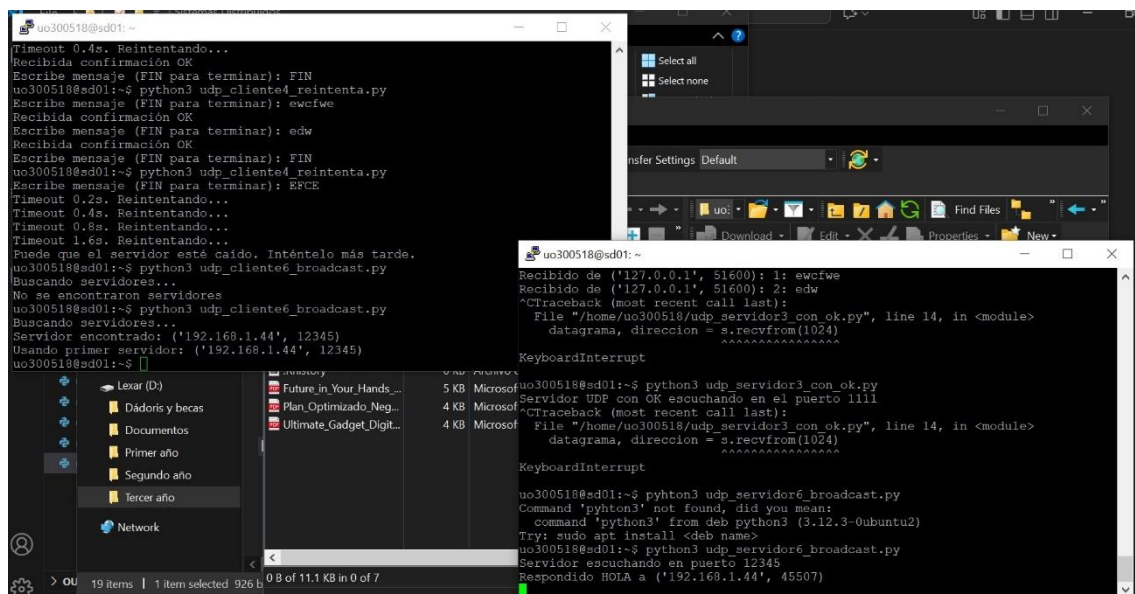
Este ejercicio permitió implementar un sistema de comunicación más fiable sobre UDP:

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



```
uo300518@sd01: ~  
Timeout 0.4s. Reintentando...  
Recibida confirmación OK  
Escribe mensaje (FIN para terminar): FIN  
uo300518@sd01:~$ python3 udp_cliente4_reintenta.py  
Escribe mensaje (FIN para terminar): ewcfe  
Recibida confirmación OK  
Escribe mensaje (FIN para terminar): edw  
Recibida confirmación OK  
Escribe mensaje (FIN para terminar): FIN  
uo300518@sd01:~$ python3 udp_cliente4_reintenta.py  
Escribe mensaje (FIN para terminar): EFCE  
Timeout 0.2s. Reintentando...  
Timeout 0.4s. Reintentando...  
Timeout 0.6s. Reintentando...  
Timeout 1.6s. Reintentando...  
Puede que el servidor esté caído. Inténtelo más tarde.  
uo300518@sd01:~$ python3 udp_cliente6_broadcast.py  
Buscando servidores...  
No se encontraron servidores  
uo300518@sd01:~$ python3 udp_cliente6_broadcast.py  
Buscando servidores...  
Servidor encontrado: ('192.168.1.44', 12345)  
Usando primer servidor: ('192.168.1.44', 12345)  
uo300518@sd01:~$  
uo300518@sd01:~$ python3 udp_servidor3_con_ok.py  
Servidor UDP con OK escuchando en el puerto 1111  
^C  
Traceback (most recent call last):  
  File "/home/uo300518/udp_servidor3_con_ok.py", line 14, in <module>  
    datagrama, direccion = s.recvfrom(1024)  
KeyboardInterrupt  
uo300518@sd01:~$ python3 udp_servidor6_broadcast.py  
Command 'python3' not found, did you mean:  
  command 'python' from deb python3 (3.12.3-0ubuntu2)  
Try: sudo apt install <deb name>  
uo300518@sd01:~$ python3 udp_servidor6_broadcast.py  
Servidor escuchando en puerto 12345  
Respondido HOLA a ('192.168.1.44', 45507)
```

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

```
uo300148@02:~/pls/S3_presencial/Ejer_7$ docker network ls
NETWORK ID          NAME                DRIVER             SCOPE
3bae3d8d5259        bridge             bridge             local
a776fedc3b94        host               host               local
297ea3055307        none              null               local
205b3165ed63        pruebas           bridge            local
uo300148@02:~/pls/S3_presencial/Ejer_7$ docker network inspect pruebas | grep Subnet -n
1:      "Subnet": "172.18.0.0/16",
uo300148@02:~/pls/S3_presencial/Ejer_7$ docker run -d --name servidor1 --network pruebas -v "$PWD":/app -w /app \
python:3.7-bullseye python3 udp_servidor6_broadcast.py
ca29013c98a2f553bd194d037811a9b76ccfca88e3694175c4f016553abb85d7
uo300148@02:~/pls/S3_presencial/Ejer_7$ docker run -d --name servidor2 --network pruebas -v "$PWD":/app -w /app \
python:3.7-bullseye python3 udp_servidor6_broadcast.py
aef359c591f26f36cc6c6901ad02b93a22cad97d0593dabb4ee3128be7cf81
uo300148@02:~/pls/S3_presencial/Ejer_7$ docker run -d --name servidor3 --network pruebas -v "$PWD":/app -w /app \
python:3.7-bullseye python3 udp_servidor6_broadcast.py
01d1511cae172d3e872ad95b2d7c460a1d76033b519a0502056318efab977a56
uo300148@02:~/pls/S3_presencial/Ejer_7$
```

```
Unpacking libatmi:amd64 (1:2.5.1-4) ...
Selecting previously unselected package libpam-cap:amd64.
Preparing to unpack .../7-libpam-cap_1:3a2.44-1+deb11u1_amd64.deb ...
Unpacking libpam-cap:amd64 (1:2.44-1+deb11u1) ...
Setting up libatmi:amd64 (1:2.5.1-4) ...
Setting up libcap2:amd64 (1:2.44-1+deb11u1) ...
Setting up libcap2-bin (1:2.44-1+deb11u1) ...
Setting up libmm10:amd64 (1:0.4-3) ...
Setting up libttables12:amd64 (1:8.7-1) ...
Setting up libbpf0:amd64 (1:0.3-2+deb11u1) ...
Setting up libpam-cap:amd64 (1:2.44-1+deb11u1) ...
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based frontend cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog.pm line 78.)
debconf: falling back to frontend: Readline
Setting up iproute2 (5.10.0-4) ...
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based frontend cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog.pm line 78.)
debconf: falling back to frontend: Readline
Processing triggers for libc-bin (2.31-13+deb11u6) ...
root@ca29013c98a2:/app# ip addr show eth0
2: eth0@if27: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 92:aa:9a:09:3a:03 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.18.0.2/16 brd 172.18.255.255 scope global eth0
        valid_lft forever preferred_lft forever
root@ca29013c98a2:/app#
```

```
uo300148@02:~/pls/S3_presencial/Ejer_7$ docker run -it --rm --name cliente --network pruebas \
-v "$PWD":/app -w /app \
python:3.7-bullseye python3 udp_cliente6_broadcast.py 172.18.255.255 9999
Servidor encontrado: ('172.18.0.4', 12345) -> SERVIDOR ACTIVO
uo300148@02:~/pls/S3_presencial/Ejer_7$
```

```
uo300148@02:~/pls/S3_presencial/Ejer_7$ docker rm -f servidor1 servidor2 servidor3 cliente
servidor1
servidor2
servidor3
Error response from daemon: No such container: cliente
uo300148@02:~/pls/S3_presencial/Ejer_7$ docker network rm pruebas
pruebas
uo300148@02:~/pls/S3_presencial/Ejer_7$
```

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.

Se creó 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 pruebas
```

Parte 2:

EJ-1: Servidor y cliente TCP con envío secuencial

```
uo300148@02:~/pls$ cd S4_online/  
uo300148@02:~/pls/S4_online$ python3 tcp_servidor1_simple.py  
Escuchando en puerto 9999 (mensajes de 5 bytes)  
Esperando un cliente..  
Conectado desde ('127.0.0.1', 43640)  
Recibido bloque: ABCDE  
Recibido bloque: ABCDE  
Recibido bloque: ABCDE  
Recibido bloque: ABCDE  
Recibido bloque: ABCDE  
Recibido FINAL → cierro sesión con cliente  
Esperando un cliente..
```

```
OSError: [Errno 98] Address already in use  
uo300148@02:~/pls/S4_online$ python3 tcp_cliente1_simple.py  
uo300148@02:~/pls/S4_online$ python3 tcp_cliente1_simple.py  
uo300148@02:~/pls/S4_online$ python3 tcp_cliente1_simple.py  
uo300148@02:~/pls/S4_online$
```

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*.

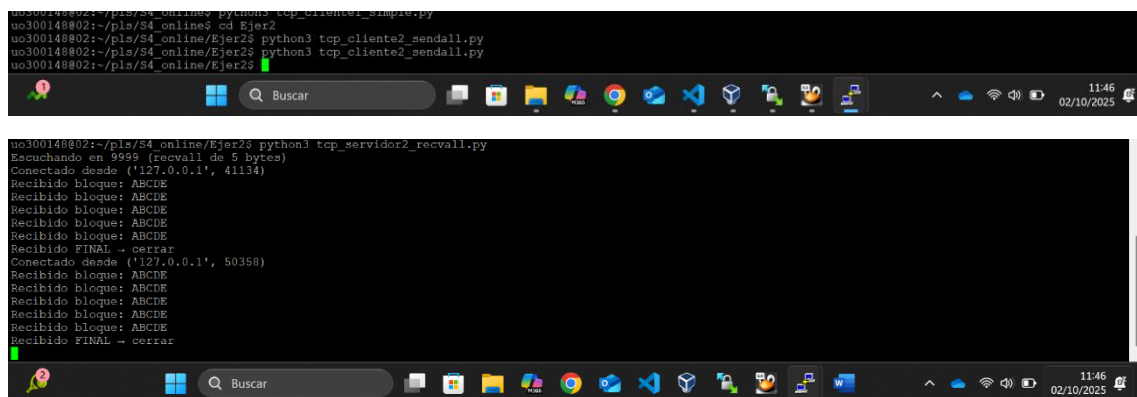
Secuencia de recepción: 5× "ABCDE" + "FINAL".

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



```
uo300148@02:~/pls/S4_online$ python3 tcp_cliente_sendall.py
uo300148@02:~/pls/S4_online/Ejer2$ python3 tcp_cliente2_sendall.py
uo300148@02:~/pls/S4_online/Ejer2$

uo300148@02:~/pls/S4_online/Ejer2$ python3 tcp_servidor2_recvall.py
Escuchando en 9999 (recvall de 5 bytes)
Conectado desde ('127.0.0.1', 41134)
Recibido bloque: ABCDE
Recibido bloque: ABCDE
Recibido bloque: ABCDE
Recibido bloque: ABCDE
Recibido bloque: ABCDE
Recibido FINAL -> cerrar
Conectado desde ('127.0.0.1', 50358)
Recibido bloque: ABCDE
Recibido bloque: ABCDE
Recibido bloque: ABCDE
Recibido bloque: ABCDE
Recibido bloque: ABCDE
Recibido FINAL -> cerrar
```

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 la 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

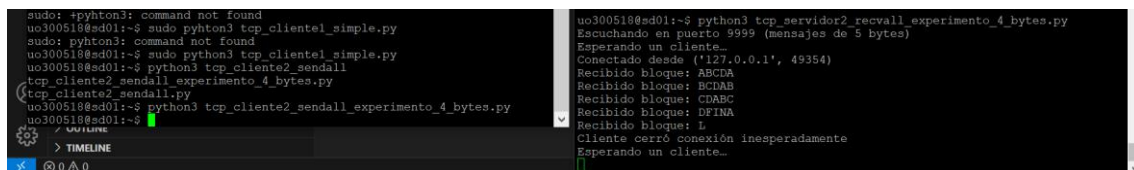
Consola del servidor mostrando conexión, recepción de bloques, cierre por "FINAL" y retorno al estado de espera.

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



```
sudo: !python3: command not found
uo300518@sd01:~$ sudo python3 tcp_cliente_simple.py
sudo: python3: command not found
uo300518@sd01:~$ sudo python3 tcp_cliente2_sendall.py
tcp_cliente2_sendall_experimento_4_bytes.py
tcp_cliente2_sendall.py
uo300518@sd01:~$ python3 tcp_cliente2_sendall_experimento_4_bytes.py
uo300518@sd01:~$

uo300518@sd01:~$ python3 tcp_servidor2_recvall_experimento_4_bytes.py
Escuchando en puerto 9999 (mensajes de 5 bytes)
Esperando un cliente_
Conectado desde ('127.0.0.1', 49354)
Recibido bloque: ABCDA
Recibido bloque: BCDAB
Recibido bloque: CDABC
Recibido bloque: DFINA
Recibido bloque: E
Cliente cerró conexión inesperadamente
Esperando un cliente_
```

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()`: "ABCD" (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.

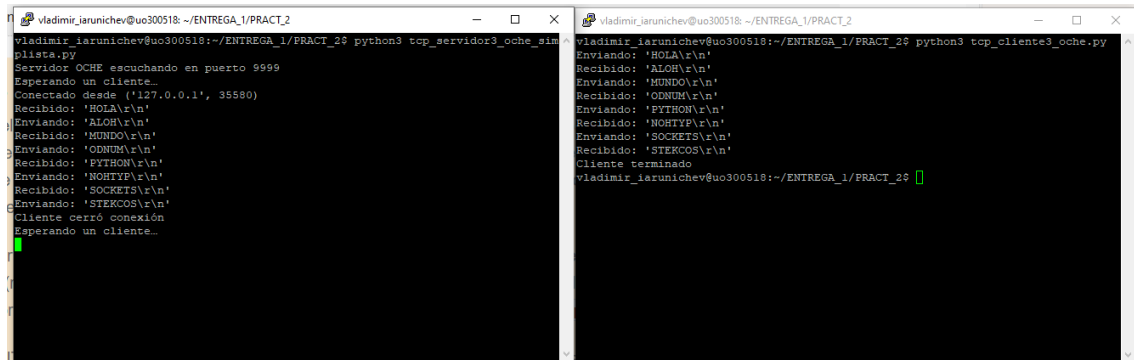
Pérdida de sincronización entre cliente y servidor.

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



```
vladimir_iarunichev@uo300518: ~/ENTREGA_1/PRACT_2
vladimir_iarunichev@uo300518:~/ENTREGA_1/PRACT_2$ python3 tcp_servidor3_oche_simplista.py
Servidor OCHE escuchando en puerto 9999
Esperando un cliente...
Conectado desde ('127.0.0.1', 35580)
Recibido: 'HOLA\r\n'
Enviando: 'ALOH\r\n'
Recibido: 'MUNDO\r\n'
Enviando: 'ODNUM\r\n'
Recibido: 'PYTHON\r\n'
Enviando: 'NOHTYP\r\n'
Recibido: 'SOCKETS\r\n'
Enviando: 'STECOS\r\n'
Cliente cerró conexión
Esperando un cliente...

vladimir_iarunichev@uo300518: ~/ENTREGA_1/PRACT_2$ python3 tcp_cliente3_oche.py
Enviando: 'HOLA\r\n'
Recibido: 'ALOH\r\n'
Enviando: 'MUNDO\r\n'
Recibido: 'ODNUM\r\n'
Enviando: 'PYTHON\r\n'
Recibido: 'NOHTYP\r\n'
Enviando: 'SOCKETS\r\n'
Recibido: 'STECOS\r\n'
Cliente terminado
vladimir_iarunichev@uo300518:~/ENTREGA_1/PRACT_2$
```

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: 'TEXTOR\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

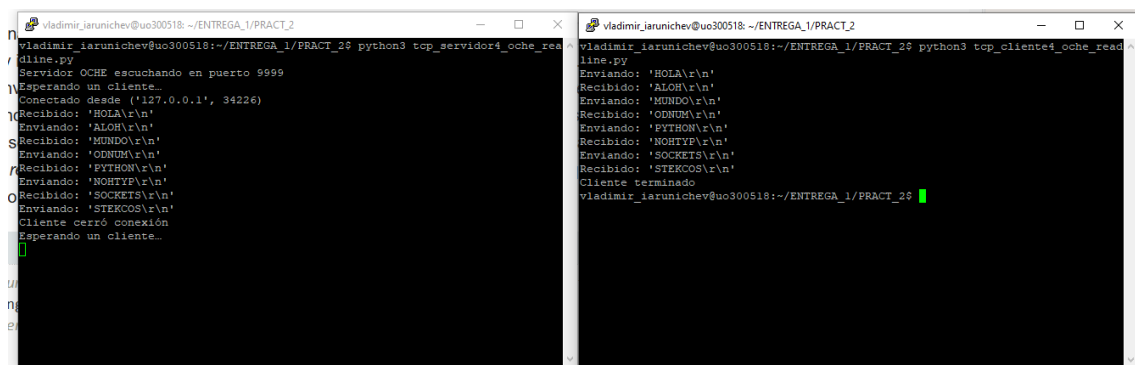
Consola del servidor mostrando la recepción y envío de mensajes invertidos.

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



```
vladimir_iarunichev@uo300518: ~/ENTREGA_1/PRACT_2
python3 tcp_servidor4_ochereadline.py
Servidor OCHE escuchando en puerto 9999
Esperando un cliente...
Conectado desde ('127.0.0.1', 34226)
Recibido: 'HOLA\r\n'
Enviando: 'ALOH\r\n'
Recibido: 'MUNDO\r\n'
Enviando: 'ODNUM\r\n'
Recibido: 'PYTHON\r\n'
Enviando: 'NOHTYP\r\n'
Recibido: 'SOCKETS\r\n'
Enviando: 'STEKCOS\r\n'
Cliente cerró conexión
Esperando un cliente...

vladimir_iarunichev@uo300518: ~/ENTREGA_1/PRACT_2
python3 tcp_cliente4_ochereadline.py
Enviando: 'HOLA\r\n'
Recibido: 'ALOH\r\n'
Enviando: 'MUNDO\r\n'
Recibido: 'ODNUM\r\n'
Enviando: 'PYTHON\r\n'
Recibido: 'NOHTYP\r\n'
Enviando: 'SOCKETS\r\n'
Recibido: 'STEKCOS\r\n'
Cliente terminado
vladimir_iarunichev@uo300518: ~/ENTREGA_1/PRACT_2
```

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_ochereadline.py` y `tcp_cliente4_ochereadline.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

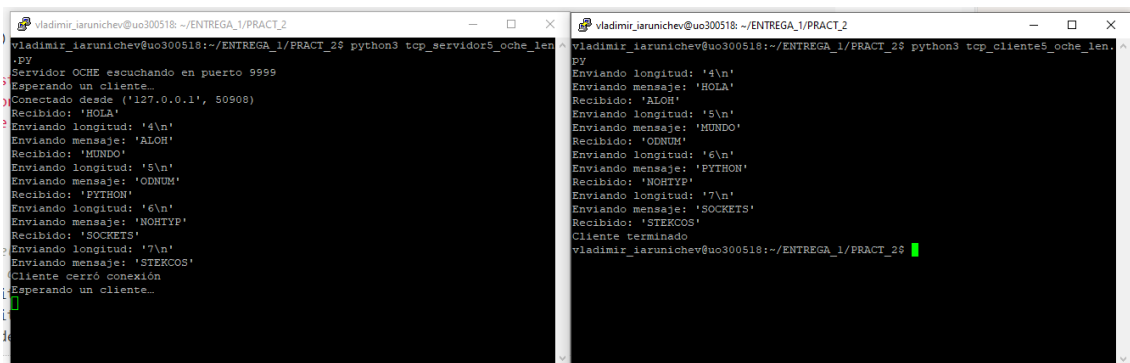
Consola del servidor mostrando la recepción ordenada de múltiples mensajes.

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



```
vladimir_iarunichev@uo300518: ~/ENTREGA_1/PRACT_2
python3 tcp_servidor5_oche_len.py
Servidor OCHE escuchando en puerto 9999
Esperando un cliente...
Conectado desde ('127.0.0.1', 50908)
Recibido: 'HOLA'
Enviando longitud: '4\n'
Enviando mensaje: 'ALOH'
Recibido: 'MUNDO'
Enviando longitud: '5\n'
Enviando mensaje: 'ODNUH'
Recibido: 'PYTHON'
Enviando longitud: '6\n'
Enviando mensaje: 'NOHTYP'
Recibido: 'SOCKETS'
Enviando longitud: '7\n'
Enviando mensaje: 'SFERCOS'
Cliente cerró conexión
Esperando un cliente...

vladimir_iarunichev@uo300518: ~/ENTREGA_1/PRACT_2
python3 tcp_cliente5_oche_len.py
Enviando longitud: '4\n'
Enviando mensaje: 'HOLA'
Recibido: 'ALOH'
Enviando longitud: '5\n'
Enviando mensaje: 'MUNDO'
Recibido: 'ODNUH'
Enviando longitud: '6\n'
Enviando mensaje: 'PYTHON'
Recibido: 'NOHTYP'
Enviando longitud: '7\n'
Enviando mensaje: 'SOCKETS'
Recibido: 'SFERCOS'
Cliente terminado
vladimir_iarunichev@uo300518: ~/ENTREGA_1/PRACT_2
```

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

```
vladimir_iarunichev@uo300518: ~/ENTREGA_1/PRACT_2
vladimir_iarunichev@uo300518:~/ENTREGA_1/PRACT_2$ ^C
vladimir_iarunichev@uo300518:~/ENTREGA_1/PRACT_2$ docker run -d --rm -p 9090:9999 --network pruebas --name servidor_oché -v $(pwd):/app python:3.7 python3 /app/tcp_servidor4_oché_readline.py
55f70d7184e6 python:3.7 "python3 /app/tcp_se..." 5 seconds ago Up 5 second
s 0.0.0.0:9090->9999/tcp, [::]:9090->9999/tcp servidor_oché
7fc59bc4eb2e python:3.7 "python /app/tcp_ser..." 4 minutes ago Up 4 minute
s servidor1
vladimir_iarunichev@uo300518:~/ENTREGA_1/PRACT_2$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
PORTS
55f70d7184e6   python:3.7     "python3 /app/tcp_se..." 5 seconds ago  Up 5 second
s 0.0.0.0:9090->9999/tcp, [::]:9090->9999/tcp  servidor_oché
7fc59bc4eb2e   python:3.7     "python /app/tcp_ser..." 4 minutes ago  Up 4 minute
s  servidor1
vladimir_iarunichev@uo300518:~/ENTREGA_1/PRACT_2$ python3 tcp_cliente4_oché_readline.py localhost 9090
Enviando: 'HOLA\r\n'
Recibido: 'ALOH\r\n'
Enviando: 'MUNDO\r\n'
Recibido: 'ODNUM\r\n'
Enviando: 'PYTHON\r\n'
Recibido: 'NOHTYP\r\n'
Enviando: 'SOCKETS\r\n'
Recibido: 'STEKCOS\r\n'
Cliente terminado
vladimir_iarunichev@uo300518:~/ENTREGA_1/PRACT_2$
```

```
uo300148@02: ~/Entega_1/p2
Se han calculado 5 de los 18 números primos solicitados
La hora actual es: 16:38:11.638045
Se han calculado 10 de los 18 números primos solicitados
La hora actual es: 16:38:40.090030
Se han calculado 15 de los 18 números primos solicitados
La hora actual es: 16:39:05.041066
Se han calculado 18 de los 18 números primos solicitados
Primos: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61]
FIN
uo300148@02:~/Entega_1/p3/1$ cd ..
uo300148@02:~/Entega_1/p3$ cd ..
uo300148@02:~/Entega_1$ cd p2
uo300148@02:~/Entega_1/p2$ python3 tcp_cliente4_oché_readline.py 192.168.201.122 9090
Enviando: 'HOLA\r\n'
Recibido: 'ALOH\r\n'
Enviando: 'MUNDO\r\n'
Recibido: 'ODNUM\r\n'
Enviando: 'PYTHON\r\n'
Recibido: 'NOHTYP\r\n'
Enviando: 'SOCKETS\r\n'
Recibido: 'STEKCOS\r\n'
Cliente terminado
uo300148@02:~/Entega_1/p2$
```

1. Objetivo

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

2. Entorno

VM Linux de prácticas con Docker instalado.

Archivos utilizados: tcp_servidor4_oche_readline.py y tcp_cliente4_oche_readline.py.

3. 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.

4. 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.

5. Evidencias (capturas propuestas)

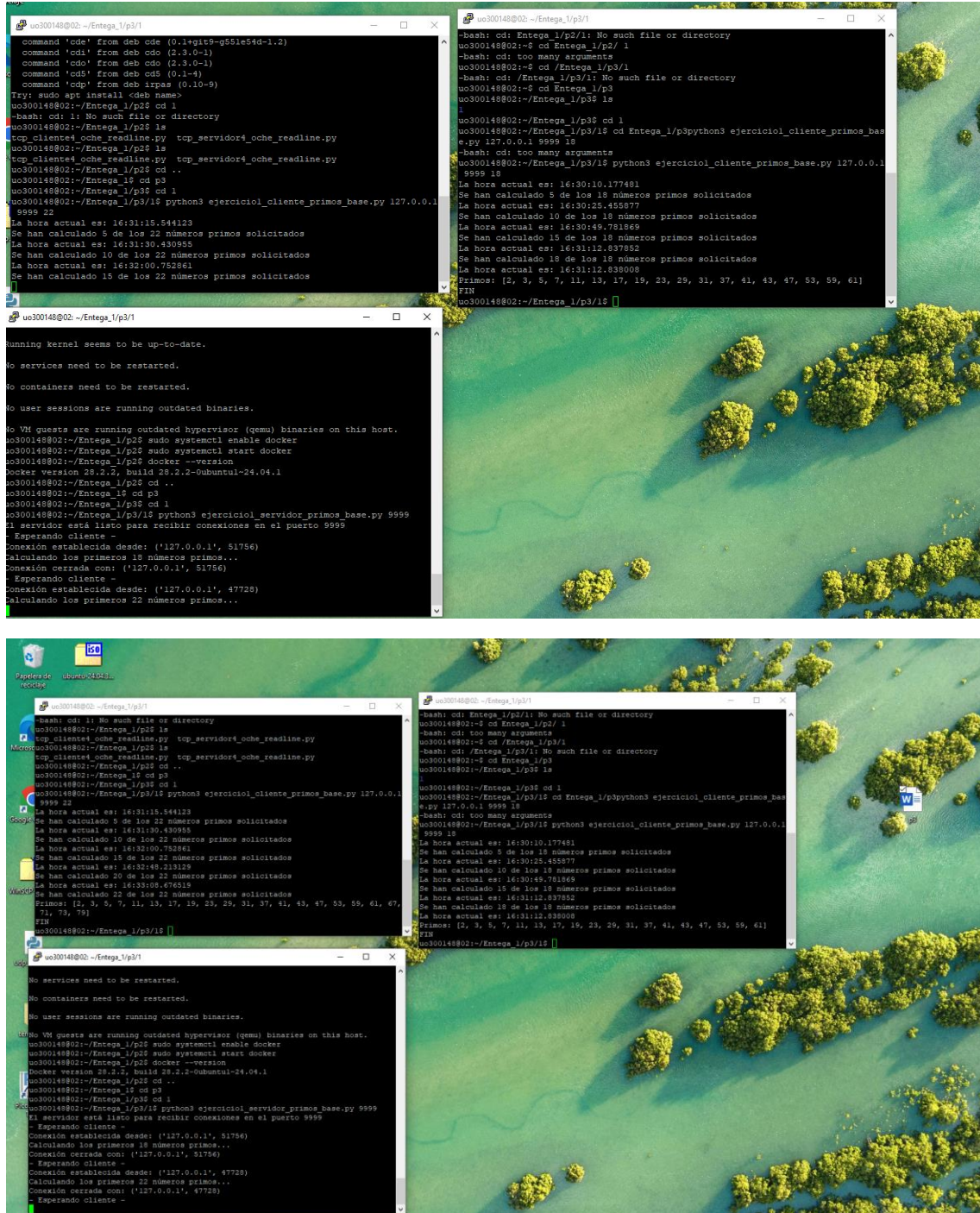
- Salida de docker ps mostrando mapeo de puertos.
- Consola de servidor en contenedor mostrando conexiones entrantes.
- Consola de cliente externo mostrando comunicación exitosa.

6. 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.

Parte 3:

EJ-1: Servidor TCP para calcular números primos



The image displays four terminal windows from a virtual machine named 'u0300148@02' with the working directory set to '/Entega_1/p3/1'. The top-left window shows the installation of dependencies: 'git', 'cdo', 'cd5', and 'lisp', followed by the installation of 'python3'. The top-right window shows the execution of 'python3 ejercicio1_cliente_primos_base.py 127.0.0.1 9999', which starts a client that sends requests to a server. The bottom-left window shows the execution of 'python3 ejercicio1_servidor_primos_base.py 9999', which starts a server listening on port 9999. The bottom-right window shows the output of the client, displaying the current time and the list of the first 22 prime numbers: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61].

```
command 'cdo' from deb cdo (0.1+git9-g551e54d-1.2)
command 'cdo' from deb cdo (2.3.0-1)
command 'cdo' from deb cdo (2.3.0-1)
command 'cd5' from deb cd5 (0.1-4)
command 'lisp' from deb lisp (0.10-9)
Try: sudo apt install <deb name>
u0300148@02:~/Entega_1/p3/1$ cd 1
-bash: cd: 1: No such file or directory
u0300148@02:~/Entega_1/p3/1$ cd 1
tcp_cliente4_ocht_readline.py tcp_servidor4_ocht_readline.py
u0300148@02:~/Entega_1/p3/1$ cd 1
tcp_cliente4_ocht_readline.py tcp_servidor4_ocht_readline.py
u0300148@02:~/Entega_1/p3/1$ cd 1
u0300148@02:~/Entega_1/p3/1$ python3 ejercicio1_cliente_primos_base.py 127.0.0.1 9999
La hora actual es: 16:31:15.544123
Se han calculado 5 de los 22 números primos solicitados
La hora actual es: 16:31:30.430955
Se han calculado 10 de los 22 números primos solicitados
La hora actual es: 16:31:00.752821
Se han calculado 15 de los 22 números primos solicitados
La hora actual es: 16:30:10.177481
Se han calculado 5 de los 18 números primos solicitados
La hora actual es: 16:30:25.455977
Se han calculado 10 de los 18 números primos solicitados
La hora actual es: 16:30:49.781869
Se han calculado 15 de los 18 números primos solicitados
La hora actual es: 16:31:12.837852
Se han calculado 18 de los 18 números primos solicitados
La hora actual es: 16:31:12.838008
Primos: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61]
FIN
u0300148@02:~/Entega_1/p3/1$

Running kernel seems to be up-to-date.
No services need to be restarted.
No containers need to be restarted.
No user sessions are running outdated binaries.
No VM guests are running outdated hypervisor (qemu) binaries on this host.
u0300148@02:~/Entega_1/p3/1$ sudo systemctl enable docker
u0300148@02:~/Entega_1/p3/1$ sudo systemctl start docker
u0300148@02:~/Entega_1/p3/1$ docker --version
Docker version 20.2.2, build 20.2.2-Ubuntu-24.04.1
u0300148@02:~/Entega_1/p3/1$ cd 1
u0300148@02:~/Entega_1/p3/1$ cd 1
u0300148@02:~/Entega_1/p3/1$ cd 1
u0300148@02:~/Entega_1/p3/1$ python3 ejercicio1_servidor_primos_base.py 9999
El servidor está listo para recibir conexiones en el puerto 9999
- Esperando cliente -
Conexión establecida desde: ('127.0.0.1', 51756)
Calculando los primeros 18 números primos...
Conexión cerrada con: ('127.0.0.1', 51756)
- Esperando cliente -
Conexión establecida desde: ('127.0.0.1', 47729)
Calculando los primeros 22 números primos...
Conexión cerrada con: ('127.0.0.1', 47729)
- Esperando cliente -
```

1. Objetivo

Diseñar una aplicación cliente-servidor TCP donde el servidor calcule los X primeros números primos solicitados por el cliente, enviando actualizaciones cada 5 números calculados.

2. Entorno

Máquina virtual Linux con Python instalado.

Archivos utilizados: `ejercicio1_servidor_primos_base.py` y `ejercicio1_cliente_primos_base.py`.

3. Procedimiento realizado

Crear los scripts de cliente y servidor con la lógica de cálculo de primos.

Lanzar el servidor en una terminal.

Lanzar dos clientes en paralelo desde otras terminales, solicitando 22 y 18 números primos respectivamente.

Observar el orden de atención de los clientes.

4. Resultados observados

El servidor atiende a los clientes de forma secuencial.

El segundo cliente no comienza a recibir datos hasta que el primero ha terminado.

Se observa una ejecución bloqueante por cliente.

Evidencias (capturas propuestas)

Consola del servidor mostrando conexión con el primer cliente y luego con el segundo.

Consolas de los clientes mostrando mensajes de progreso y recepción de lista final.

5. Conclusión

La implementación secuencial del servidor provoca que los clientes sean atendidos uno tras otro. No hay concurrencia ni paralelismo, lo que genera tiempos de espera elevados para múltiples clientes.

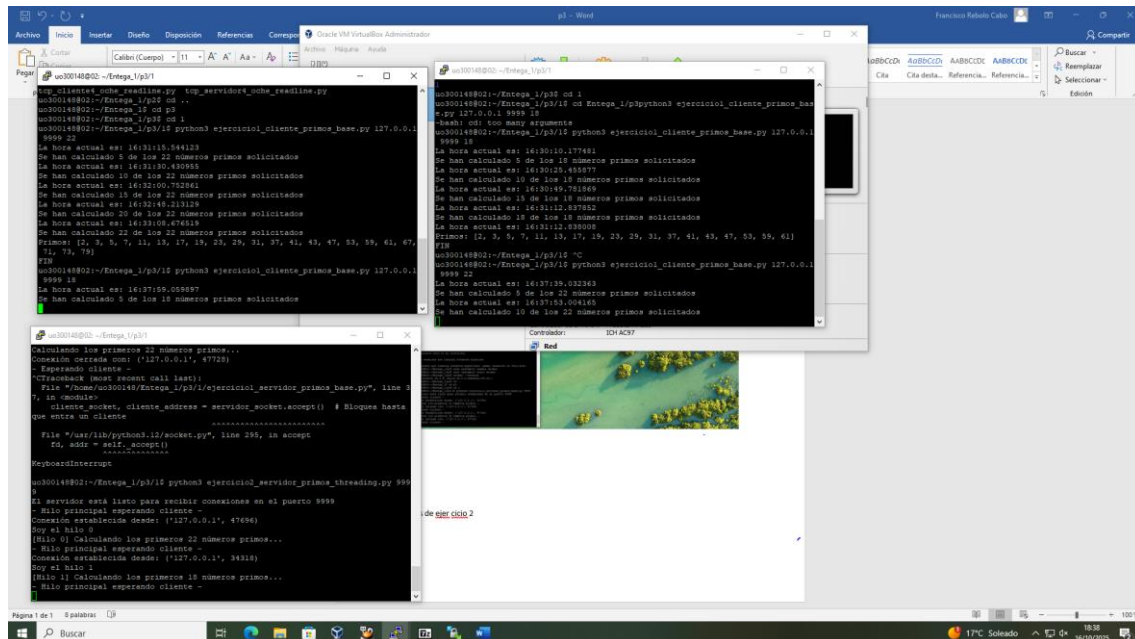
6. Qué hace:

- El servidor calcula primos y envía un mensaje cada 5 primos:
`Se han calculado N de los X números primos solicitados`
- Al terminar, manda la lista completa `Primos: [...]` y luego `FIN`.
- El cliente imprime cada mensaje junto con la hora local y se cierra al recibir `FIN`.

7. Respuesta conceptual:

- Con este servidor “base” (iterativo, un solo hilo), atiende a un cliente cada vez.
- El segundo cliente empieza a ser procesado cuando el primero termina (secuencial).
- No hay concurrencia, ni paralelismo, ni asincronía en el servidor base; sólo un bucle `accept()` → procesa → cierra → vuelve a `accept()`.

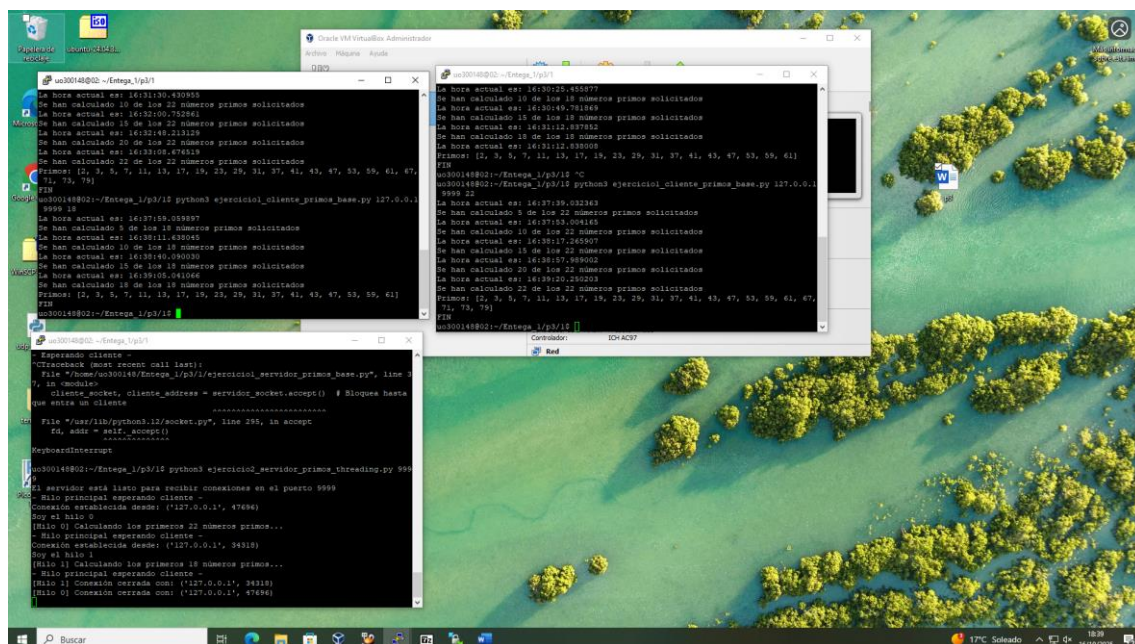
EJ-2: Programación concurrente con select()



```
python3 ejercicio2_servidor_primos_base.py 127.0.0.1 9999
La hora actual es: 16:31:15.544123
Se han calculado 5 de los 22 números primos solicitados
La hora actual es: 16:31:30.430955
Se han calculado 10 de los 22 números primos solicitados
La hora actual es: 16:32:00.752841
Se han calculado 15 de los 22 números primos solicitados
La hora actual es: 16:32:44.213129
Se han calculado 20 de los 22 números primos solicitados
La hora actual es: 16:33:05.676519
Se han calculado 22 de los 22 números primos solicitados
Primos: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79]
FIN

python3 ejercicio2_cliente_primos_base.py 127.0.0.1 9999
La hora actual es: 16:30:10.177461
Se han calculado 5 de los 18 números primos solicitados
La hora actual es: 16:30:25.455577
Se han calculado 10 de los 18 números primos solicitados
La hora actual es: 16:30:40.731849
Se han calculado 15 de los 18 números primos solicitados
La hora actual es: 16:31:12.397882
Se han calculado 18 de los 18 números primos solicitados
La hora actual es: 16:31:12.388008
Primos: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61]
FIN

python3 ejercicio2_servidor_primos_threading.py 9999
El servidor está listo para recibir conexiones en el puerto 9999
Milo principal esperando cliente -
Conexión establecida desde: ("127.0.0.1", 47694)
Soy el hilo 0
[Milo 0] Calculando los primeros 22 números primos...
Milo principal esperando cliente -
Conexión establecida desde: ("127.0.0.1", 34318)
Soy el hilo 1
[Milo 1] Calculando los primeros 18 números primos...
Milo principal esperando cliente -
Conexión establecida desde: ("127.0.0.1", 34318)
Milo 0] Conexión cerrada con: ("127.0.0.1", 47694)
```



```
python3 ejercicio2_servidor_primos_base.py 127.0.0.1 9999
La hora actual es: 16:32:00.752841
Se han calculado 10 de los 22 números primos solicitados
La hora actual es: 16:32:44.213129
Se han calculado 15 de los 22 números primos solicitados
La hora actual es: 16:33:05.676519
Se han calculado 20 de los 22 números primos solicitados
La hora actual es: 16:33:05.676519
Se han calculado 22 de los 22 números primos solicitados
Primos: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79]
FIN

python3 ejercicio2_cliente_primos_base.py 127.0.0.1 9999
La hora actual es: 16:30:10.177461
Se han calculado 5 de los 18 números primos solicitados
La hora actual es: 16:30:25.455577
Se han calculado 10 de los 18 números primos solicitados
La hora actual es: 16:30:40.731849
Se han calculado 15 de los 18 números primos solicitados
La hora actual es: 16:31:12.397882
Se han calculado 18 de los 18 números primos solicitados
La hora actual es: 16:31:12.388008
Primos: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61]
FIN

python3 ejercicio2_servidor_primos_threading.py 9999
El servidor está listo para recibir conexiones en el puerto 9999
Milo principal esperando cliente -
Conexión establecida desde: ("127.0.0.1", 47694)
Soy el hilo 0
[Milo 0] Calculando los primeros 22 números primos...
Milo principal esperando cliente -
Conexión establecida desde: ("127.0.0.1", 34318)
Soy el hilo 1
[Milo 1] Calculando los primeros 18 números primos...
Milo principal esperando cliente -
Conexión establecida desde: ("127.0.0.1", 34318)
Milo 0] Conexión cerrada con: ("127.0.0.1", 47694)
```

1. Objetivo

Modificar el servidor para que pueda atender múltiples clientes de forma concurrente usando la función `select()`.

2. Entorno

Máquina virtual Linux con Python.

Archivo utilizado: `ejercicio2_servidor_primos_select.py`.

3. Procedimiento realizado

Implementar el servidor con `select()` para monitorizar múltiples sockets.

Lanzar el servidor y dos clientes simultáneamente.

Observar cómo se intercalan las respuestas del servidor.

Verás que **ambos clientes reciben progreso en paralelo** porque cada conexión se atiende en su propio hilo (`threading.Thread`).

4. Resultados observados

El servidor atiende a varios clientes de forma intercalada.

Cada cliente recibe actualizaciones sin esperar a que otro finalice.

Se logra concurrencia aparente en un solo hilo de ejecución.

Evidencias (capturas propuestas)

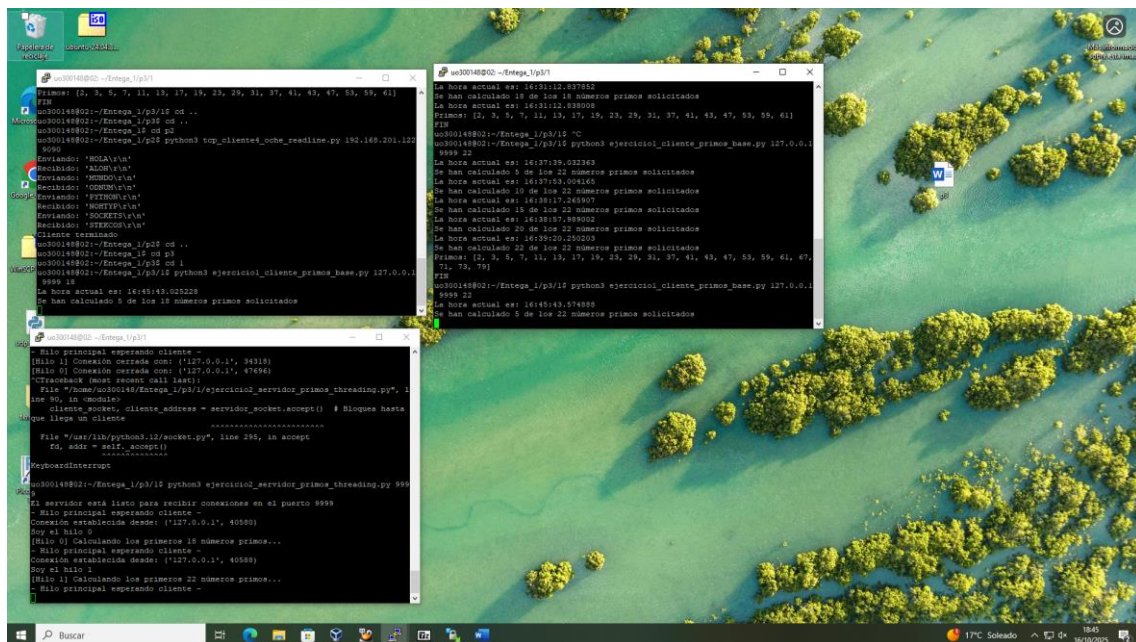
Consola del servidor mostrando múltiples sockets activos.

Consolas de los clientes recibiendo mensajes de progreso simultáneamente.

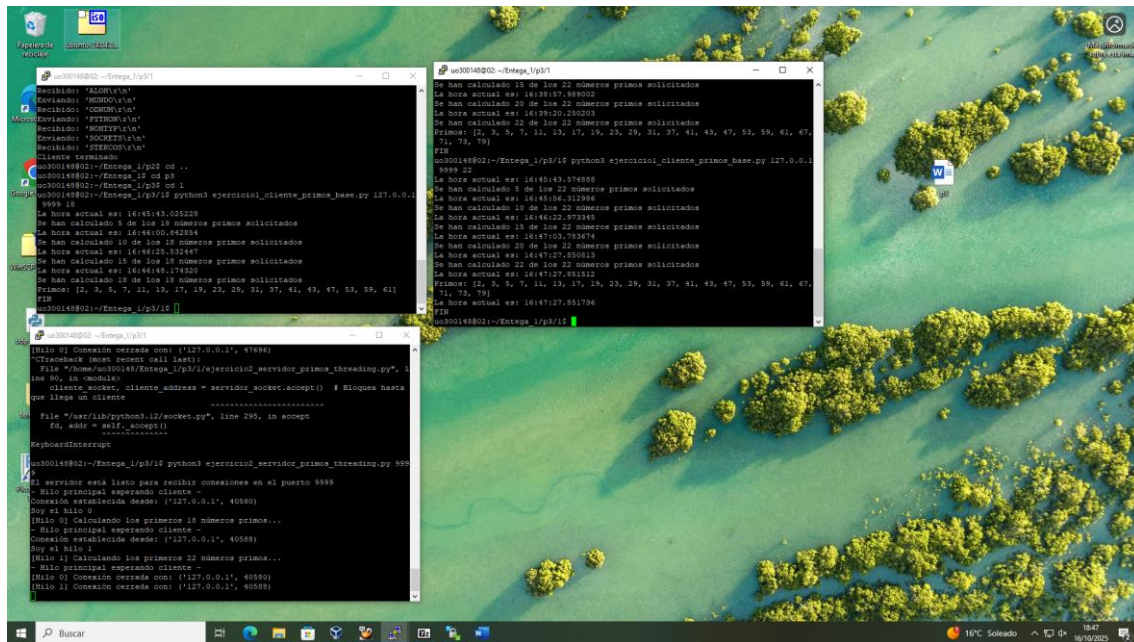
5. Conclusión

El uso de `select()` permite una ejecución concurrente en un solo hilo, mejorando la experiencia del cliente. Aunque no hay paralelismo real, se reduce el tiempo de espera percibido.

EJ-3: Programación paralela con procesos (fork)



```
python3 top_client_001.py 127.0.0.1 9999
La hora actual es: 16:11:12.037852
Se han calculado 19 de los 22 números primos solicitados
La hora actual es: 16:11:12.038008
Primos: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61]
python3 ejercicio02_cliente_primos_base.py 127.0.0.1 9999
La hora actual es: 16:11:12.032363
Se han calculado 5 de los 22 números primos solicitados
La hora actual es: 16:11:12.034165
Se han calculado 10 de los 22 números primos solicitados
La hora actual es: 16:11:12.045907
Se han calculado 15 de los 22 números primos solicitados
La hora actual es: 16:11:12.049005
Se han calculado 20 de los 22 números primos solicitados
La hora actual es: 16:11:12.050203
Se han calculado 22 de los 22 números primos solicitados
Primos: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79]
python3 ejercicio02_servidor_primos_threading.py 9999
El servidor está listo para recibir conexiones en el puerto 9999
Hilo principal esperando cliente -
[Main 0] Conexión cerrada con: ('127.0.0.1', 34315)
[Main 0] Conexión cerrada con: ('127.0.0.1', 47694)
Traceback (most recent call last):
  File "C:\Users\user\AppData\Local\Programs\Python\Python311\python.exe", line 11, in <module>
    client_socket, client_address = servidor_socket.accept() # Bloquea hasta
    que llegue un cliente
  File "C:\Users\user\AppData\Local\Programs\Python\Python311\python.exe", line 11, in <module>
    client_socket, client_address = servidor_socket.accept() # Bloquea hasta
    que llegue un cliente
KeyboardInterrupt
```



1. Objetivo

Adaptar el servidor para que utilice procesos en lugar de hilos, permitiendo atención simultánea a múltiples clientes.

2. Entorno

Máquina virtual Linux con Python.

Archivo utilizado: ejercicio3_servidor_primos_fork.py.

3. Procedimiento realizado

Implementar el servidor con `os.fork()` para crear procesos hijos por cliente.

Lanzar el servidor y dos clientes simultáneamente.

Verificar que cada cliente es atendido por un proceso independiente.

4. Resultados observados

El servidor crea un proceso hijo por cada cliente.

Los clientes son atendidos en paralelo, sin interferencias.

Cada proceso cierra los sockets que no necesita, evitando fugas.

Evidencias (capturas propuestas)

Consola del servidor mostrando creación de procesos hijos.

Consolas de los clientes recibiendo respuestas en paralelo.

Salida de `ps` mostrando procesos activos.

El **proceso padre** se queda en `accept()` y por **cada conexión** hace `os.fork()`.

El **proceso hijo** cierra el socket pasivo, atiende al cliente completo (envía progreso cada 5 primos y luego la lista + `FIN`) y **termina**.

El **padre** cierra el socket de datos y vuelve a aceptar, permitiendo **múltiples clientes en paralelo** aprovechando varios cores.

5. Conclusión

La programación paralela con procesos permite una atención simultánea real a múltiples clientes. Es útil para aislar tareas intensivas, aunque consume más recursos que los hilos.