1. Introducción

Hoy quiero mostrarles un ejemplo práctico de algo que, si bien no es parte obligatoria de los contenidos del curso, vale la pena entender: un servidor de red construido en C usando **sockets**.

Sé que ustedes ya vienen trabajando con `fork`, `pipes` y otros mecanismos del sistema operativo. La mayoría está en pleno desarrollo del TP, y han visto cómo se pueden comunicar procesos entre sí. Pero ahora vamos a dar un paso más allá y ver cómo se construye algo que se usa todos los días en la vida real: un servidor de red que puede atender múltiples conexiones, como lo hacen los servidores web, de archivos o incluso de videojuegos online.

¿Por qué les muestro esto si no entra en el examen? Porque este ejemplo reúne muchos de los conceptos que ya aprendieron y les da un contexto real. Les va a permitir ver cómo `fork`, `read`, `write` y los `file descriptors` que venimos usando aparecen también en escenarios mucho más grandes y útiles. No se trata de un ejercicio de laboratorio o un juguete académico: esto es **cómo funciona Internet por debajo**.

Además, como ya tienen una buena base, van a ver que no necesitamos conocimientos avanzados de redes ni saber TCP a fondo. Con lo que ya manejan, pueden entender cómo funciona la estructura de un servidor y por qué los mecanismos de Unix son tan potentes. De hecho, cuando yo era ayudante en Taller de Programación, explicábamos algo muy parecido a esto y los estudiantes lo entendían y lo implementaban sin problema, incluso sin haber cursado la materia de Redes.

Así que si alguna vez pensaron que `fork` o los `pipes` eran cosas que sólo se usan en prácticas de laboratorio, hoy van a ver que no es así. Vamos a trabajar con un ejemplo real, claro, y muy cercano a lo que sucede en un servidor de verdad.

2. ¿Qué es un servidor y qué es un cliente?

Antes de meternos con el código, quiero que repasemos juntos una idea muy sencilla pero fundamental: la diferencia entre un **servidor** y un **cliente**. A esta altura, la mayoría de ustedes ya sabe más o menos cómo funciona Internet. Nacemos, nos entregan un iPad y ya estamos conectados. Pero vale la pena detenerse un momento a ponerle nombres precisos a las cosas.

La **definición oficial** es simple:

- Un **servidor** es una aplicación que **espera conexiones** de otros programas.
- Un **cliente** es el que **inicia** esas conexiones.

Pensemos en un ejemplo clásico: un navegador web y un servidor web. El navegador (el cliente) quiere acceder a una página. Entonces, se conecta al servidor (por ejemplo, el que aloja Wikipedia) y le pide la información. El servidor recibe esa conexión y responde con los datos que el cliente pidió: texto, imágenes, videos, etc.

Lo mismo pasa en un videojuego online: vos, desde tu computadora o consola, actuás como cliente. Te conectás al servidor del juego, que está escuchando conexiones de miles de jugadores al mismo tiempo.

En todos esos casos, el flujo básico es el mismo:

1. El servidor está quietito, escuchando.
2. El cliente decide iniciar la conversación.
3. El servidor acepta esa conexión y la atiende.

Y esto que parece tan abstracto o avanzado lo vamos a poder implementar usando exactamente las herramientas que ya conocen. Vamos a construir un servidor que escucha conexiones y un cliente que se conecta a él, usando `fork`, `read`, `write`, y una pequeña porción nueva: `sockets`.

Aunque las conexiones sean de red, para el sistema operativo no hay tanta diferencia. Lo que nos interesa hoy es **la estructura básica del servidor**. Y como van a ver enseguida, no se trata de magia: con unas pocas líneas de código bien pensadas, podemos hacer que nuestro programa escuche conexiones como lo haría cualquier servidor real.

3. Estructura de un servidor secuencial

Vamos ahora al ejemplo concreto. Acá les muestro un servidor básico escrito en C que escucha conexiones en la red y les responde con un simple "Hola mundo". Parece sencillo —y lo es—, pero este ejemplo nos permite entender muchos conceptos fundamentales del sistema operativo aplicados en un contexto real.

3.1. La función `main`

El corazón de este servidor está en el `main`, que sigue una estructura muy clara:

1. Llama a `create_server_socket()` para preparar el socket y ponerlo en modo de escucha en el puerto 8080.
2. Entra en un bucle infinito (`for (;;)`) donde:

    - Acepta una nueva conexión con `accept()`.
    - Imprime un mensaje indicando que llegó una nueva conexión.
    - Usa `write()` para enviar el mensaje `"Hola mundo\n"` al cliente.
    - Cierra la conexión con `close()`.

Hasta acá, nada muy diferente a lo que ya vieron cuando trabajaron con pipes o con archivos. La diferencia es que, en lugar de leer o escribir en un archivo, lo estamos haciendo a través de una conexión de red. Pero la interfaz es exactamente la misma: `write()` y `close()` funcionan como siempre.

3.2. La función `create_server_socket`

Esta función es una abstracción que introducimos nosotros y que describimos por completitud de la explicación. Encapsula toda la lógica necesaria para crear un socket de servidor:

- Primero llama a `socket()` para obtener un file descriptor que represente el socket. Este descriptor no es un archivo en disco, pero sí sigue la misma lógica: es un número entero que representa una fuente o destino de datos.
- Configura la estructura `sockaddr_in` con la dirección IP y el puerto donde queremos que escuche. En este caso, `INADDR_ANY` indica que va a aceptar conexiones desde cualquier IP, y el puerto es el 8080.
- Luego hace `bind()` para asociar ese socket con la dirección y el puerto configurados.
- Por último, llama a `listen()` para poner el socket en estado de escucha, listo para aceptar conexiones entrantes.

Si algo falla en cualquiera de estos pasos, la función llama a `err_sys()`, que imprime el error y termina el programa.

3.3. La llamada a `accept()` (y su parecido con `open()`)

En el corazón del bucle principal del servidor tenemos la llamada a `accept()`. Esta función es fundamental en cualquier servidor de red, y su comportamiento tiene un paralelismo muy interesante con otra función que ya conocen muy bien: `open()`.

Recordemos qué hace `open()` en el caso de archivos: uno le pasa una ruta (por ejemplo, `"archivo.txt"`) y el sistema operativo devuelve un **file descriptor** que representa ese archivo abierto. Con ese descriptor luego podemos leer o escribir usando `read()` o `write()`. Y, por supuesto, debemos cerrarlo cuando terminamos, usando `close()`.

`accept()` funciona de forma análoga, pero en lugar de abrir un archivo, **abre una conexión** de red. Cuando alguien se conecta al servidor, `accept()` acepta esa conexión y devuelve un **nuevo file descriptor**, diferente del que se obtuvo originalmente con `socket()`. Ese nuevo descriptor representa una conexión específica entre el servidor y un cliente.

Podemos pensar entonces que:

- El socket en modo escucha es como un "archivo especial" que está esperando a que alguien lo abra.
- La llamada a `accept()` es como `open()`: crea una instancia concreta de acceso (en este caso, una conexión de red).
- Y una vez que tenemos el descriptor devuelto por `accept()`, podemos usar `write()`, `read()` y `close()` como si estuviéramos manejando un archivo.

Esta comparación no es casual. Es parte del diseño deliberado de Unix, que busca que todas las interfaces de entrada/salida (archivos, pipes, sockets) se manejen de la misma forma: mediante **file descriptors** y un conjunto común de llamadas (`read`, `write`, `close`). Esta unificación simplifica muchísimo la programación a bajo nivel, y es una de las razones por las que trabajar con sockets no es tan complicado como parece a primera vista.

3.4. Por qué es importante cerrar las conexiones

Cada vez que aceptamos una nueva conexión, el sistema nos da un nuevo file descriptor. Si no los cerramos, eventualmente vamos a llenar la tabla de descriptores del proceso, y eso puede generar errores difíciles de detectar. Por eso, cada vez que terminamos de manejar una conexión, hacemos `close(connfd)` para liberar ese recurso.

En este servidor, ese cierre ocurre justo después de enviar el mensaje al cliente. El servidor vuelve entonces al comienzo del bucle, listo para aceptar otra conexión.

3.5. Un servidor secuencial

A esta altura ya pueden ver que este servidor es **secuencial**: maneja una sola conexión a la vez. Si un cliente se conecta y el servidor está ocupado con otro, ese cliente tendrá que esperar. En muchos casos reales, eso no es aceptable. Pero este diseño básico nos sirve como primer paso.

En resumen: este servidor escucha, acepta, responde y cierra, todo en un único flujo de control. Es simple, directo, y nos permite ver que los sockets en Unix siguen el mismo modelo de `read`, `write` y `close` que los pipes o los archivos. Una gran victoria del diseño simple y coherente de Unix.

4. Cliente simple

Ya entendimos cómo se comporta el servidor: escucha conexiones, acepta una, manda un mensaje y cierra. Ahora, para cerrar el ciclo, veamos brevemente cómo funciona el **cliente** que se conecta a este servidor.

El cliente es un programa separado, y su único objetivo es conectarse al servidor y leer lo que este le mande. En este ejemplo, el servidor está escuchando en el **host local** (`localhost`, o IP `127.0.0.1`) y en el **puerto 8080**. Entonces el cliente establece la conexión con esos datos y hace lo siguiente:

1. **Establece una conexión** con el servidor mediante un socket.
2. **Llama a `read()`** sobre ese socket para recibir los datos.
3. **Imprime lo recibido**, en este caso el mensaje `"Hola mundo\n"`.
4. **Cierra la conexión** con `close()`.

Hasta ahí parece todo muy directo. Sin embargo, hay un pequeño detalle técnico que vale la pena destacar: **la llamada a `read()` sobre un socket no siempre devuelve todos los datos que el servidor envió**.

4.1. Lecturas parciales en sockets

Cuando trabajamos con archivos locales, solemos asumir que si pedimos leer 100 bytes, `read()` nos va a devolver 100 bytes (salvo que estemos al final del archivo). Pero con sockets la historia es distinta.

¿Por qué? Porque los datos vienen por la red, y la red no es un medio perfectamente confiable ni instantáneo. Puede haber demoras, congestión o simplemente cortes en la transmisión. Entonces, si el servidor manda 20 bytes, puede pasar que `read()` nos devuelva, por ejemplo, solo 10. Los otros 10 quizás lleguen en la siguiente llamada.

Por eso, cuando trabajamos con sockets, lo correcto es usar un **bucle `while`** para seguir leyendo hasta que:

- Se hayan recibido todos los datos esperados, o
- La llamada a `read()` devuelva **cero** (`0`), lo cual indica que **el otro extremo cerró la conexión**.

Ese valor cero tiene un significado especial. Nos está diciendo: *"ya no hay más datos, y no va a haber más, porque el otro lado hizo un `close()`"*. Esto nos muestra que `close()` no solo libera recursos, sino que también tiene un efecto semántico sobre la comunicación: le dice al otro extremo que terminó de enviar datos.

4.2. Decodificar los bytes

Otro punto a considerar es que los datos que el cliente recibe son simplemente **bytes**. En este caso sabemos que representan texto, pero el sistema no lo sabe. Por eso, el cliente tiene que decidir cómo interpretarlos. Si estamos mandando un mensaje como `"Hola mundo\n"`, no hay mucho misterio. Pero si estuviéramos enviando datos más complejos —por ejemplo, un archivo JSON o binario— el cliente tendría que saber cómo decodificarlos correctamente. Por ejemplo pueden ser caracteres en Unicode y soportar símbolos de otros idiomas. El alumno interesado puede leer este post.

Esa interpretación queda del lado de la aplicación. El sistema operativo no se mete. Solo garantiza que los bytes lleguen en el mismo orden en el que fueron enviados, y eso es una de las maravillas del diseño de los sockets sobre TCP.

5. Limitaciones del servidor secuencial

Hasta ahora, el servidor que construimos funciona correctamente: escucha conexiones, atiende una a la vez, y responde con un mensaje. Sin embargo, hay una **limitación importante** en este diseño que vale la pena entender bien: **el servidor secuencial sólo puede atender una conexión por vez**.

5.1. ¿Qué pasa si una conexión tarda mucho?

Imaginemos un caso práctico: cambiamos nuestro servidor para que, en lugar de enviar simplemente `"Hola mundo"`, envíe un archivo de 2 GB. ¿Qué pasaría?

Bueno, el servidor recibiría la conexión, abriría el archivo, y empezaría a enviarlo por el socket. Pero mientras está ocupado en esa tarea —leyendo pedacitos del archivo con `read()` y enviándolos con `write()`— no puede hacer nada más. No puede aceptar nuevas conexiones, porque el flujo de ejecución está completamente bloqueado en esa única tarea.

Esto es un **cuello de botella**: cualquier cliente que intente conectarse mientras el servidor está ocupado se va a quedar esperando. Técnicamente, lo que ocurre es que el sistema operativo mantiene una **cola de conexiones pendientes**, donde los clientes van a esperar a que en el proceso que inició la escucha se llame accept(). En tanto no se llame a esta función, permanecen a la espera.

En nuestro ejemplo, mientras el servidor está enviando un archivo, está atrapado en un bucle de `write()` y no puede volver a `accept()` para atender nuevas conexiones.

5.2. `accept()` y las llamadas bloqueantes

Este es un buen momento para introducir un concepto muy importante: **las llamadas bloqueantes**.

Cuando el servidor llama a `accept()`, el programa **se bloquea** hasta que llegue una nueva conexión. Lo mismo ocurre con `read()`: se bloquea hasta que haya datos disponibles. Esto no es un error; es parte del diseño. En muchos casos, ese bloqueo es útil porque simplifica el flujo del programa.

El problema aparece cuando esas llamadas bloqueantes impiden que se realicen otras tareas.

5.3. ¿Qué podemos hacer?

La pregunta natural es: *¿cómo hacemos para que el servidor pueda aceptar nuevas conexiones mientras atiende una que tarda?*

La respuesta más directa, usando lo que ya aprendimos, es: **usar `fork()`**. La idea es que el proceso padre se quede haciendo `accept()` todo el tiempo, y por cada conexión nueva que llega, cree un **proceso hijo** que se encargue de atenderla.

De esa forma, el padre queda libre para seguir aceptando nuevas conexiones sin bloquearse, mientras los hijos se encargan de procesar y responder cada una de ellas.

Este enfoque nos lleva a la siguiente evolución natural del servidor: **el servidor concurrente**, basado en `fork()`. Es el modelo que vamos a explorar en la próxima sección, y como verán, es sorprendentemente fácil de implementar con las herramientas que ya manejan.

6. Introducción al servidor concurrente con `fork()`

En la sección anterior vimos cómo un servidor secuencial tiene una limitación importante: no puede atender más de una conexión a la vez. La solución que propongo ahora es sencilla y poderosa: **usar `fork()` para crear procesos hijos**, y así lograr que el servidor se vuelva concurrente.

6.1. La idea general

La idea es dividir el trabajo: el **proceso padre** se dedica exclusivamente a aceptar conexiones, y por cada una que llega, **crea un proceso hijo** para que se encargue de atenderla. De esta forma, el servidor puede seguir escuchando nuevas conexiones sin bloquearse mientras otro proceso está ocupado respondiendo a un cliente.

Este patrón —padre que acepta, hijos que atienden— es muy común y se usa en servidores reales, como por ejemplo **PostgreSQL**. Muchos otros usan hilos (`threads`), que vamos a ver más adelante. Pero como hasta ahora trabajamos con `fork()`, este modelo es perfecto para ilustrar el concepto sin introducir nuevas herramientas.

6.2. Análisis del código

Veamos cómo se implementa este servidor concurrente:

El bucle `for(;;)`

Este bucle infinito es la estructura central del servidor. En cada iteración:

1. El servidor llama a `accept()` y obtiene un nuevo socket (`connfd`).
2. Luego hace un `fork()`:
    - Si estamos en el **hijo** (`pid == 0`):
        - Imprimimos el PID del hijo.
        - Cerramos el socket de escucha (`listenfd`), porque el hijo no lo necesita.
        - Usamos `write()` para enviar el mensaje al cliente.
        - Cerramos el socket de conexión.
        - Llamamos a `return 0` para salir del hijo y evitar que continúe ejecutando el bucle del padre.
    - Si estamos en el **padre**:
        - Cerramos el socket de conexión (`connfd`), porque lo manejará el hijo.

        - El padre vuelve al inicio del bucle y se queda esperando nuevas conexiones.

    - Si ocurre un error en `fork()`, se imprime un mensaje de error.

Importancia del `return` en el hijo

Una de las trampas comunes es olvidar el `return` (o `exit`) en el proceso hijo. Si se omite, el hijo terminará ejecutando también el bucle principal, generando más `fork()`s innecesarios. Eso puede llevar rápidamente a una **explosión de procesos**, haciendo que el sistema se vuelva inestable o inutilizable. Así que: ¡siempre terminar explícitamente el proceso hijo!

6.3. El resultado: un servidor concurrente

Con este esquema, ahora tenemos un **servidor verdaderamente concurrente**. Cada vez que un cliente se conecta, se crea un proceso nuevo que maneja la conexión de forma independiente. El padre, mientras tanto, sigue aceptando otras conexiones.

Este tipo de arquitectura tiene muchas ventajas:

- Permite aprovechar sistemas con múltiples núcleos.
- Aísla las conexiones entre sí: si un hijo falla, no afecta a los demás.
- Es sencillo de implementar con `fork()`.

7. Detalles del manejo de procesos

En el servidor concurrente que vimos recién, cada conexión que llega genera un nuevo proceso hijo. Esa solución es poderosa y flexible, pero trae aparejado un nuevo compromiso: **tenemos que encargarnos de los procesos hijos una vez que terminan**.

Esto nos introduce al concepto de **zombie processes** y al uso de `wait()`.

7.1. ¿Qué es un proceso zombie?

Cuando un proceso hijo termina su ejecución (porque llegó a un `return`, un `exit`, o simplemente finalizó el `main()`), el sistema operativo **no lo elimina de inmediato**. En lugar de eso, mantiene cierta información sobre ese proceso en una tabla interna. ¿Por qué? Porque **el padre podría querer saber cómo terminó ese hijo**, por ejemplo si devolvió un código de error.

Ese estado intermedio, donde el proceso ya no está vivo ni corriendo, pero **tampoco fue completamente eliminado**, se llama **estado zombie**. Y sí, no es una metáfora: ese es el nombre oficial.

Un proceso zombie no consume CPU, pero **sí consume recursos del sistema**, especialmente entradas en la tabla de procesos. Si no los limpiamos correctamente, pueden acumularse y agotar esos recursos.

7.2. El rol de `wait()`

Para evitar que los procesos hijos queden como zombies, el padre debe hacer una llamada a la función `wait()` o `waitpid()`. Estas funciones permiten:

- Obtener el código de salida del hijo.
- Notificar al sistema operativo que ya no es necesario guardar información sobre ese hijo.
- Liberar completamente sus recursos.

En nuestro caso, el padre podría quedar bloqueado si llama a `wait()` directamente. Como no queremos que el padre deje de aceptar conexiones, usamos una solución elegante: **el manejador de señales**.

7.3. Manejando hijos con señales

No vamos a ahondar demasiado en el tema de señales. Pero podemos entender que las señales son otra forma en que un proceso puede comunicarse con otro.

En el código del servidor concurrente, se registra un manejador para la señal `SIGCHLD`:

```
signal(SIGCHLD, sigchld_handler);
```

Esto significa que, cada vez que un hijo termina, el sistema le envía esa señal al padre. En respuesta, el padre ejecuta la función `sigchld_handler()`, que contiene lo siguiente:

```
int pid;
while ((pid = waitpid(-1, NULL, WNOHANG)) > 0){
    printf("Hijo con PID %d terminó.\n", pid);
}
```

Esta función intenta hacer `waitpid()` sobre cualquier hijo (`-1` indica que no importa cuál), y con la opción `WNOHANG`, que le indica que **no se bloquee si no hay hijos terminados**. Si hay alguno, hace el `waitpid()` y libera los recursos. Repite esto en un bucle por si terminaron varios hijos al mismo tiempo.

Este patrón es común y efectivo. Permite que el servidor continúe aceptando conexiones mientras se asegura de que **los hijos no queden como zombies**.

7.4. ¿Y si el padre muere antes que el hijo?

Este es otro caso muy interesante. Supongamos que un proceso crea un hijo con `fork()`, pero el **padre termina antes** que el hijo. En ese caso, el hijo queda **huérfano**.

Pero en Linux (y en Unix en general), **todos los procesos deben tener un padre**. Si el padre original muere, el sistema operativo **re-asigna automáticamente** el hijo a un proceso especial: el **proceso 1**.

Históricamente este proceso fue `init`, hoy suele ser `systemd`, dependiendo de la distribución. Pero cumple la misma función: **adoptar procesos huérfanos**.

Y lo más importante: este proceso 1 **siempre hace `wait()`** por sus hijos. Así que aunque el padre original no esté más, el sistema garantiza que los hijos eventualmente serán recolectados correctamente y **no quedarán como zombies permanentes**.

Esto quiere decir que en Unix **no existen procesos realmente huérfanos**. El sistema operativo siempre resuelve la situación y se asegura de que todos los procesos tengan un padre válido.

8. Ejecución y demostración del servidor concurrente

Una vez que tenemos todo el código armado, es hora de ver cómo se comporta nuestro servidor en la práctica. Esta parte es clave para terminar de entender cómo interactúan los distintos componentes que construimos.

8.1. Arrancando el servidor

Cuando compilamos y ejecutamos el servidor concurrente, lo primero que veremos en pantalla es:

```
listening connections...
```

Ese mensaje proviene de la función `create_server_socket()` y nos indica que el socket de escucha fue correctamente creado, enlazado al puerto 8080, y puesto en modo de escucha. A partir de ese momento, el servidor queda listo para aceptar conexiones entrantes.

8.2. Ejecutando el cliente

Desde otra terminal, podemos ejecutar el cliente que se conecta al puerto 8080 del localhost. Apenas lo hacemos, en la salida del servidor veremos algo así como:

```
Hijo con PID: 12345 manejando conexión
Hijo con PID 12345 terminó.
```

Estas dos líneas son fundamentales:

- La primera indica que el proceso hijo fue creado con éxito, y que está manejando la conexión con el cliente.
- La segunda indica que el hijo terminó su trabajo (en este caso, enviar `"Hola mundo"` y cerrar la conexión), y que fue correctamente recolectado por el manejador de señales.

Mientras tanto, en la consola del cliente, veremos que recibió el mensaje:

```
Hola mundo
```

Eso confirma que la conexión se estableció, que el servidor envió los datos correctamente, y que el cliente los leyó sin problemas.

8.3. ¿Cómo sabe el hijo cuál es su PID?

En el mensaje del servidor aparece el número de PID del hijo. Para que el proceso hijo lo sepa, usamos la system call `getpid()`:

```
printf("Hijo con PID: %d manejando conexión\n", getpid());
```

Esto puede parecer trivial, pero tiene mucho valor para depurar o monitorear procesos en sistemas reales. Recordemos que, en el caso del padre, `fork()` devuelve el PID del hijo. Pero en el hijo, `fork()` siempre devuelve cero, así que si queremos conocer su propio PID, debemos llamarlo explícitamente.

8.4. Servidor verdaderamente concurrente

Este patrón se vuelve más evidente si lanzamos múltiples clientes rápidamente. Cada uno va a disparar un nuevo proceso hijo, que se encarga de enviar el mensaje de forma independiente. El padre, mientras tanto, sigue aceptando conexiones. Así es como logramos concurrencia real: **varios procesos ejecutándose al mismo tiempo, cada uno atendiendo a un cliente distinto**.

Esta es la estructura que muchos servidores reales han utilizado históricamente. Y aunque hoy en día muchos servidores modernos usan `threads`, eventos o incluso `async/await` en lenguajes de alto nivel (eg. Javascript), el modelo basado en `fork()` sigue siendo válido, y entenderlo te da las bases para comprender todo lo demás.

9. Reflexión sobre la abstracción de sockets

Una de las mayores virtudes que tiene este ejemplo —y que me parece importante subrayar— es cómo **los sockets se integran naturalmente con todo lo que ya aprendimos** sobre archivos, pipes y file descriptors. A pesar de que estamos trabajando con redes, que en principio parecen un tema mucho más complejo, las herramientas que usamos son las mismas de siempre: `read()`, `write()`, `close()`.

Y eso no es casualidad. Es una de las grandes decisiones de diseño que hicieron Thompson y Ritchie cuando crearon Unix.

9.1. Todo es un stream de bytes

En Unix, casi todo lo que puede leerse o escribirse se representa como un **file descriptor**. Eso incluye:

- Archivos en disco
- Pipes entre procesos
- Dispositivos
- Y, como vemos ahora, **sockets de red**

En todos los casos, usamos la misma interfaz: `write()` para enviar datos, `read()` para recibirlos, y `close()` para liberar el recurso. ¿Qué significa esto? Que no necesitamos aprender una API nueva para cada tipo de comunicación. El sistema operativo se encarga de las diferencias internas, y nosotros programamos siempre con las mismas primitivas.

9.2. `write()` escribe bytes, y nada más

Es importante entender que `write()` no tiene ninguna noción de “mensajes”, “estructuras” ni “tipos”. Solo escribe una **secuencia de bytes**.

En nuestro ejemplo, cuando enviamos `"Hola mundo\n"`, lo hacemos con:

```
write(connfd, buff, strlen(buff));
```

Ese buffer (`buff`) es un puntero a `char`, pero en realidad eso significa: “una tira de bytes”. El servidor no sabe ni le importa si esos bytes representan texto, un número, una imagen o un video.

De hecho, esto es lo mismo que pasa en aplicaciones reales. Cuando una aplicación web manda un objeto JSON, lo que hace en el fondo es **serializarlo como bytes**, y luego mandarlo como bytes por el socket.

El `write()` que estamos usando ahora podría mandar:

- Un mensaje de texto
- Un archivo de audio
- Un paquete codificado en formato binario
- O cualquier otra cosa

Todo lo que necesitamos es que esos datos se puedan representar como bytes. A ese proceso se lo llama **serialización**.

9.3. Interpretar los bytes: tarea del receptor

Del lado del cliente, lo que se recibe son también bytes. Es la **aplicación** la que debe decidir cómo interpretarlos. Si el cliente espera un texto, puede mostrarlos como tal. Si espera un archivo binario, puede guardarlos directamente. El sistema operativo no interviene en esa interpretación.

Tampoco existe una separación entre “mensajes” dentro de un flujo de bytes. Si el servidor quiere mandar varios mensajes, tiene que definir una convención para separar uno de otro (por ejemplo, usando un carácter especial o indicando la longitud antes de cada bloque).

En resumen: **la capa de red ofrece un flujo continuo de bytes, no de estructuras**. Y eso nos da total flexibilidad, pero también la responsabilidad de organizar los datos correctamente en el nivel de aplicación.

9.4. La potencia de una buena abstracción

Esta forma de trabajar es una muestra de lo potente que es la abstracción de Unix. En lugar de ofrecer cientos de interfaces distintas para cada tipo de comunicación, el sistema operativo nos da una sola, bien diseñada y uniforme. Y esa uniformidad hace que podamos escribir servidores de red usando exactamente los mismos conceptos que usamos para leer un archivo o comunicar dos procesos con un pipe.

Como ingenieros de software, entender esta base nos permite:

- **Valorar las capas de abstracción que usamos todos los días** (como HTTP, JSON, WebSockets, etc.)
- **Comprender lo que ocurre por debajo** cuando usamos librerías o frameworks modernos
- **Diagnosticar y resolver problemas a bajo nivel**, cuando las herramientas de alto nivel fallan

10. Consideraciones finales y bibliografía

Lo que vimos a lo largo de este ejemplo práctico es mucho más que un servidor de "Hola mundo". Es una ventana a **cómo funciona la comunicación entre computadoras a nivel de sistema operativo**, y cómo herramientas simples como `fork()` y `write()` nos permiten construir sistemas reales, robustos y concurrentes.

10.1. Lo que aprendimos

- Vimos que **un servidor de red no es conceptualmente tan distinto de un programa que lee o escribe archivos**. Usa las mismas llamadas del sistema, lo que demuestra la potencia del diseño unificado de Unix.
- Entendimos que **`accept()` es como `open()`**: nos da un nuevo file descriptor con el que podemos interactuar.
- Implementamos un **servidor secuencial** y luego lo transformamos en un **servidor concurrente usando `fork()`**, capaz de atender múltiples clientes simultáneamente.
- Aprendimos a **manejar procesos hijos** y evitar zombies con `wait()` y el uso de **señales**.
- Vimos cómo **la abstracción de “todo es un stream de bytes”** nos permite enviar cualquier tipo de dato a través de un socket, sin que el sistema operativo tenga que entender qué estamos mandando.

Este ejemplo también nos prepara para **entender mejor herramientas modernas**, como servidores web en Python, Node.js, o frameworks de backend en general. Todos ellos, en algún punto, dependen de estos conceptos fundamentales que Unix diseñó hace décadas y que siguen vigentes hoy.

10.2. ¿Y en la vida real?

Hoy en día, en la mayoría de los casos no vamos a escribir servidores desde cero en C. Vamos a usar librerías, frameworks o incluso servicios gestionados que abstraen todos estos detalles. Sin embargo, **entender cómo funciona la base** nos da una ventaja enorme. Nos permite comprender mejor los errores, optimizar aplicaciones, y —cuando haga falta— bajar al nivel del sistema para resolver problemas difíciles.

Además, hay servidores reales que siguen usando este mismo modelo. **PostgreSQL**, por ejemplo, crea un proceso hijo por cada conexión. Otros servidores, como **Apache** o algunos modos de **Nginx**, también siguen una arquitectura basada en procesos o hilos. No es solo historia: esto se usa todos los días.

10.3. Bibliografía recomendada

Para quienes se entusiasmen con estos temas y quieran profundizar, hay un libro clásico que recomiendo especialmente:

📘 **W. Richard Stevens –** ***Unix Network Programming***

Este libro es una referencia fundamental sobre programación de sockets en Unix. Está muy bien escrito, con ejemplos claros, y cubre desde lo básico hasta temas avanzados como multiplexación de conexiones, protocolos personalizados, y servidores de alto rendimiento. Muchas de las ideas y ejemplos que usamos acá están inspirados en ese texto.