ANÁLISIS DEL PROGRAMA programa07_servidor_grupal

1. **Seguridad por Identidad:** Aunque dos alumnos tengan el mismo puerto efímero (si vienen de máquinas distintas), el servidor los separa por IP. No hay forma de que el Alumno B "suplante" al Alumno A porque el servidor obtiene los datos directamente de la estructura del socket gestionada por el kernel (getpeername).

2. **Eficiencia con select():** Este código no usa el 100% de la CPU esperando. El proceso se "duerme" en el kernel y solo despierta cuando hay electricidad (paquetes) llegando a la tarjeta de red.

3. **Manipulación de Datos:** Utilizan inet_ntop y ntohs para que los datos binarios de la red tengan sentido para un humano.

4. Si usan nc, escriban algo y presionen Enter. Si escriben un mensaje muy largo que supere el BUFFER_SIZE, verán que el servidor lo recibe en "pedazos". Esto confirma que TCP es un **protocolo de flujo de bytes** (stream) y no de mensajes.

5. **Orden de Cierre:** ¿Qué pasa si cierro el servidor (Ctrl+C en mi terminal) mientras ustedes están conectados? Sus terminales de nc se cerrarán casi de inmediato porque el servidor les envía un paquete **FIN** a todos antes de morir. Es la demostración perfecta del *Four-way Handshake*.

El **Three-way Handshake** es para abrir la conexión y el **Four-way Handshake** es para cerrarla.

Mientras que para abrir una conexión ambos deben estar de acuerdo simultáneamente, para cerrarla TCP es muy educado y permite lo que llamamos **"Cierre Gradual"**. Como TCP es *full-duplex*, cada lado debe cerrar su parte de la conversación de forma independiente.

Los 4 pasos explicados:

Imaginemos que el **Cliente** decide que ya terminó de hablar y quiere cerrar la conexión con el **Servidor**.

1. FIN (del Cliente al Servidor)

El Cliente envía un paquete con la bandera **FIN** (Finish) activa.

Traducción: "He terminado de enviarte datos, ya no tengo nada más que decir".

Estado del Cliente: Entra en FIN-WAIT-1.

2. ACK (del Servidor al Cliente)

El Servidor recibe el FIN y responde con un **ACK** (Acknowledgment).

Traducción: "Recibido, entiendo que ya no me vas a enviar nada más. Pero espera... yo todavía podría tener datos pendientes para enviarte a ti".

Estado del Servidor: Entra en CLOSE-WAIT.

Estado del Cliente: Recibe el ACK y entra en FIN-WAIT-2.

3. FIN (del Servidor al Cliente)

Una vez que el Servidor termina de enviar cualquier dato que le quedara pendiente (por ejemplo, terminar de escribir un log o enviar un último mensaje de "Adiós"), envía su propio paquete **FIN**.

Traducción: "Yo también he terminado. Ya puedes cerrar todo".

Estado del Servidor: Entra en LAST-ACK.

4. ACK (del Cliente al Servidor)

El Cliente recibe el FIN del servidor y responde con el último **ACK**.

Traducción: "Entendido. ¡Adiós!".

Estado del Servidor: Al recibir este ACK, el socket desaparece (estado CLOSED).

Estado del Cliente: Entra en un estado especial llamado **TIME-WAIT**.

Si observan la documentación de la función select() menciona “select - synchronous I/O multiplexing”, pero cuando lanzan un cliente, el envío de mensajes es asincrono, es decir, tu cliente puede enviar un mensaje tras otro y no es necesario esperar a que el otro extremo responda. ¿Por qué?

La función select() es síncrono para tu programa (el servidor), pero TCP es asíncrono para la red.

¿Por qué dice la documentación que la función select() es síncrona?

Cuando la documentación dice "synchronous I/O multiplexing", se refiere a la llamada a la función dentro del código en C.

Bloqueo síncrono: Cuando el servidor ejecuta la línea activity = select(...), el programa se detiene por completo ahí. No pasa a la siguiente línea de código hasta que ocurre algo en la red.

Aviso síncrono: Cuando select() despierta, le entrega al programa una lista de quién tiene datos. Entonces el programa debe leer esos datos manualmente (recv). Mientras tu programa está leyendo el mensaje del Cliente A, no puede estar haciendo otra cosa; tiene que terminar esa tarea para volver a llamar a select().

Es síncrono porque el servidor sigue un orden lineal: Esperar -> Despertar -> Leer -> Procesar -> Volver a esperar.

¿Por qué el envío de mensajes parece Asíncrono?

Lo que observamos con los clientes enviando mensajes seguidos sin esperar respuesta es gracias a la arquitectura de Buffers (Memorias intermedias) del Kernel y de TCP.

Cuando el cliente envía un mensaje, este no viaja directamente "al programa" del servidor. Viaja a la cola de recepción (Recv-Q) del sistema operativo del servidor.

El cliente puede enviar 10 mensajes seguidos, por poner un ejemplo. El Kernel del servidor los recibe todos, los ordena y le manda un ACK (acuse de recibo) al cliente por cada uno. El cliente ve que el envío fue exitoso y sigue enviando.

La ilusión de asincronía: El cliente no necesita que el programa en C "lea" el mensaje para enviarle el siguiente; solo necesita que el Kernel del servidor lo reciba.

Conclusión:

select() es una función síncrona para gestionar un flujo de datos que, gracias a los buffers de TCP, se siente asíncrono y fluido para los usuarios del chat.