Informe Trabajo El API de Sockets

Integrantes

Rubén Angoso Berrocal DNI 70958754M

Óscar Hernández Hernández DNI 70918137Y

cliente.c

main(int , char const *)

Es la función principal de un programa cliente que se comunica con un servidor a través de TCP o UDP. Aquí está el desglose:

1. **Verificación de argumentos**: El programa espera recibir exactamente tres argumentos de la línea de comandos. Si no recibe tres argumentos, imprime un mensaje de uso y termina con un código de salida de 1.
2. **Verificación del nombre del archivo**: El tercer argumento debe ser uno de los siguientes nombres de archivo: "ordenes.txt", "ordenes1.txt" o "ordenes2.txt". Si no es ninguno de estos, el programa imprime un mensaje de uso y termina con un código de salida de 1.
3. **Selección del protocolo**: El segundo argumento debe ser "TCP" o "UDP". Si es "TCP", el programa llama a la función `clienteTCP` y pasa el nombre del servidor y el nombre del archivo como argumentos. Si es "UDP", el programa llama a la función `clienteUDP` y pasa el nombre del servidor y el nombre del archivo como argumentos.

En resumen, este código es la entrada principal al programa cliente. Verifica que los argumentos de la línea de comandos sean correctos y luego llama a la función correspondiente para iniciar la comunicación con el servidor a través de TCP o UDP.

handler(int)

Esta función toma un parámetro `sigNum` que representa el número de la señal recibida.

Dentro de la función, se imprime el mensaje "Cerrando los socket y saliendo" utilizando la función `printf`. Luego, se cierra un socket utilizando la función `close` y se finaliza el programa utilizando la función `exit` con un código de salida de 0.

En resumen, esta función es un manejador de señales que se encarga de cerrar un socket y finalizar el programa cuando se recibe una señal específica.

clienteTCP(char * , char *)

Se utiliza para establecer una conexión TCP con un servidor y enviar solicitudes leídas de un archivo. Aquí está el desglose:

1. **Inicialización**: La función comienza inicializando varias estructuras y variables que se utilizarán más adelante. También establece un manejador de señales para la señal SIGINT.

2. **Creación del socket**: Se crea un socket TCP utilizando la función `socket`. Si la creación del socket falla, se imprime un mensaje de error y la función retorna 1.

3. **Obtención de la dirección del servidor**: Se utiliza la función `getaddrinfo` para obtener información sobre el servidor al que se va a conectar el cliente. Si `getaddrinfo` falla, se imprime un mensaje de error, se cierra el socket y la función retorna 1.

4. **Conexión al servidor**: Se intenta conectar al servidor utilizando la función `connect`. Si la conexión falla, se imprime un mensaje de error, se cierra el socket y la función retorna 1.

5. **Apertura del archivo**: Se abre el archivo del que se leerán las solicitudes utilizando la función `fopen`. Si la apertura del archivo falla, se imprime un mensaje de error y la función retorna 1.

6. **Envío de solicitudes y recepción de respuestas**: Se lee cada línea del archivo y se envía al servidor. Luego, se recibe una respuesta del servidor. Si el envío o la recepción fallan, se imprime un mensaje de error, se cierra el socket y la función retorna 1.

7. **Cierre del servicio**: Si la respuesta del servidor es "221 Cerrando el servicio\r\n", la función retorna 0, indicando que el servicio se ha cerrado correctamente.

En resumen, esta función implementa un cliente TCP básico que se conecta a un servidor, envía solicitudes leídas de un archivo y recibe respuestas del servidor.

clienteUDP(char * , char *)

La función `clienteUDP` es similar a `clienteTCP`, pero hay algunas diferencias clave debido a las diferencias entre los protocolos TCP y UDP:

1. **Creación del socket**: En `clienteUDP`, se crea un socket UDP con `SOCK_DGRAM` en lugar de un socket TCP con `SOCK_STREAM`.

2. **Bind del socket**: En `clienteUDP`, se realiza un `bind` del socket a una dirección local antes de enviar datos. Esto no es necesario en `clienteTCP` porque `connect` se encarga de eso.

3. **Envío y recepción de datos**: En `clienteUDP`, se utilizan las funciones `sendto` y `recvfrom` para enviar y recibir datos. Estas funciones permiten especificar la dirección del destinatario para cada mensaje, lo cual es necesario en UDP porque no hay una conexión establecida. En `clienteTCP`, se utilizan las funciones `send` y `recv`, que envían y reciben datos a través de la conexión establecida.

4. **No hay conexión al servidor**: En `clienteUDP`, no hay una llamada a `connect` para establecer una conexión con el servidor. Esto es porque UDP es un protocolo sin conexión, por lo que no es necesario establecer una conexión antes de enviar y recibir datos.

En resumen, aunque `clienteUDP` y `clienteTCP` realizan tareas similares, hay diferencias importantes en cómo se implementan debido a las diferencias entre los protocolos TCP y UDP.

servidor.c

main(int , char const *)

Implementación de un servidor que puede manejar conexiones tanto TCP como UDP simultáneamente. Aquí está el desglose:

1. **Inicialización**: El servidor se inicializa generando una semilla para la generación de números aleatorios, definiendo varias variables y estructuras necesarias, y abriendo un archivo de registro.

2. **Configuración del socket TCP**: El servidor configura un socket TCP, lo asocia a una dirección y puerto específicos, y lo pone en modo de escucha. Si ocurre algún error durante este proceso, el servidor imprime un mensaje de error y termina.

3. **Configuración del socket UDP**: De manera similar, el servidor también configura un socket UDP y lo asocia a una dirección y puerto específicos.

4. **Configuración de las señales**: El servidor configura una función de manejo para la señal SIGINT, que se envía cuando se presiona Ctrl+C en la terminal.

5. **Bucle principal**: El servidor entra en un bucle infinito en el que espera a que se produzca alguna actividad en cualquiera de los sockets. Utiliza la función `select` para esto, que bloquea la ejecución hasta que se puede leer de alguno de los sockets.

6. **Manejo de conexiones TCP**: Si se detecta actividad en el socket TCP, el servidor acepta la nueva conexión y crea un nuevo proceso para manejarla. El proceso hijo recibe un mensaje del cliente, genera una respuesta utilizando la función `generarRespuesta`, y envía la respuesta al cliente. Este proceso se repite hasta que se recibe el mensaje "221 Cerrando el servicio", momento en el que se cierra la conexión.

7. **Manejo de conexiones UDP**: Si se detecta actividad en el socket UDP, el servidor lee el mensaje del cliente, genera una respuesta, y envía la respuesta al cliente. Al igual que con las conexiones TCP, este proceso se repite hasta que se recibe el mensaje "221 Cerrando el servicio".

8. **Cierre del archivo de registro**: Cuando se termina el servidor (por ejemplo, al recibir una señal SIGINT), se cierra el archivo de registro.

En resumen, este código implementa un servidor que puede manejar múltiples conexiones TCP y UDP simultáneamente, utilizando procesos hijos para manejar cada conexión individualmente. Cada conexión se maneja de acuerdo a una lógica específica definida en la función `generarRespuesta`.

generarRespuesta(char * , char *)

Genera una respuesta a un mensaje recibido en un servidor. Aquí está el desglose:

1. **Verificación del mensaje "HOLA"**: Si el mensaje recibido es "HOLA" y la bandera `flagHola` es 0, la función lee una línea aleatoria de un archivo, la divide en una pregunta y

un número, genera un número aleatorio de intentos entre 5 y 50, y genera una respuesta que incluye la pregunta y el número de intentos. Luego, establece `flagHola` a 1 y retorna.

2. **Verificación de la bandera `flagHola`**: Si `flagHola` es 0, la función genera una respuesta de "500 Error de sintaxis" y retorna.

3. **Verificación del mensaje "ADIOS"**: Si el mensaje recibido es "ADIOS" y `flagHola` es 1, la función establece `flagHola` a 0, genera una respuesta de "221 Cerrando el servicio" y retorna.

4. **Verificación del mensaje "+"**: Si el mensaje recibido es "+" y `flagHola` es 1, la función lee una nueva línea aleatoria del archivo, la divide en una pregunta y un número, genera un nuevo número aleatorio de intentos, y genera una respuesta que incluye la nueva pregunta y el número de intentos.

5. **Verificación del mensaje "RESPUESTA"**: Si el mensaje recibido comienza con "RESPUESTA" y `flagHola` es 1, la función verifica el número en el mensaje. Si los intentos son 0, genera una respuesta de "375 FALLO". Si el número es menor que el número objetivo y los intentos son mayores que 0, decrementa los intentos y genera una respuesta de "354 MAYOR" con el número de intentos. Si el número es mayor que el número objetivo y los intentos son mayores que 0, decrementa los intentos y genera una respuesta de "354 MENOR" con el número de intentos. Si el número es igual al número objetivo y los intentos son mayores que 0, genera una respuesta de "350 ACIERTO".

6. **Mensaje de error por defecto**: Si ninguna de las condiciones anteriores se cumple, la función genera una respuesta de "500 Error de sintaxis".

En resumen, esta función implementa la lógica del servidor para responder a los mensajes recibidos de un cliente. La respuesta generada depende del contenido del mensaje y del estado actual del servidor.

***leerLineaAleatoria(const char *)**

Función que lee una línea aleatoria de un archivo. Aquí está el desglose:

1. **Apertura del archivo**: La función comienza abriendo el archivo cuyo nombre se pasa como argumento. Si la apertura del archivo falla, se imprime un mensaje de error y la función retorna NULL.

2. **Conteo de líneas**: Luego, la función cuenta el número de líneas en el archivo. Esto se hace leyendo cada carácter del archivo y aumentando un contador cada vez que se encuentra un carácter de nueva línea ('\n').

3. **Generación de un número aleatorio**: Una vez que se ha contado el número de líneas, la función genera un número aleatorio entre 1 y el número de líneas.

4. **Lectura de la línea aleatoria**: La función vuelve al principio del archivo y lee cada línea hasta que llega a la línea que corresponde al número aleatorio. Esta línea se guarda en una variable y se le quita el carácter de nueva línea al final.

5. **Cierre del archivo y retorno de la línea**: Finalmente, la función cierra el archivo y retorna la línea leída.

En resumen, esta función implementa una forma de leer una línea aleatoria de un archivo en C.