

Universidad de Alicante

Práctica 2.2

TCP/IP. CLIENTE SERVIDOR

AUTOR:

Mauro Valls Vidal


CURSO: 2024/2025

2º Ingeniería en Inteligencia Artificial

Sistemas Operativos Y Distribuidos (33662)

—Servidor —

1 – Creación del Socket:



```
1 // ----- 1 SOCKET (Extremo de comunicación envío/recibo de paquetes) -----
2 servidor_fd = socket(AF_INET, SOCK_STREAM, 0);
3 if (servidor_fd == -1) {
4     perror("Error al crear el socket del servidor");
5     exit(EXIT_FAILURE);
6 }
7
```

Aquí se crea un punto de comunicación (socket) que se usará para hablar con los clientes. Se determina el protocolo de transporte.

- AF_INET = Trabajaremos con IPv4.
- SOCK_STREAM = Protocolo de transporte TCP (UDP sería SOCK_DGRAM).

* TCP = Protocolo orientado a conexión (asegura que datos lleguen y en orden) y es fiable con control de errores.

- 0 = Selecciona automáticamente protocolo por defecto.

Si no da fallos, se devuelve un número entero que representa "FILE DESCRIPTOR" del socket, y se guarda en `servidor_fd`. Será usado más adelante en funciones `bind`, `listen`, `accept`, etc.

En caso de error, se devuelve "-1", y se muestra que ha fallado y finaliza el programa.

2 - Bind (Asignar el socket al puerto):

```
1 // Configurar la dirección del servidor
2 direccion_servidor.sin_family = AF_INET;
3 direccion_servidor.sin_port = htons(PUERTO);
4 direccion_servidor.sin_addr.s_addr = INADDR_ANY;
```

Aquí el servidor asocia el socket a una dirección IP y puerto.

- **direccion_servidor.sin_family = AF_INET:**

sin_family = Especifica la familia de direcciones del socket
AF_INET = Indica que trabajamos con direcciones IPv4.

- **direccion_servidor.sin_port = htons(PUERTO):**

sin_port = Especifica el puerto al que se asociará.
htons(PUERTO) = Convierte el puerto indicado en la variable PUERTO (9999) en un formato que pueda entender la red.

- **direccion_servidor.sin_addr.s_addr = INADDR_ANY:**

sin_addr.s_addr = Especifica la dirección IP en la que el servidor estará escuchando.
INADDR_ANY = Escuchará a todas las interfaces de redes (Ethernet, WiFi).

* En caso de querer que solo escuche a una IP, sería: `inet_addr("IP")`

```
1 // ----- 2 BIND (Asociar el socket a la dirección y puerto) -----
2 if (bind(servidor_fd, (struct sockaddr *)&direccion_servidor, sizeof(direccion_servidor)) == -1) {
3     perror("Error al asociar el socket al puerto");
4     close(servidor_fd);
5     exit(EXIT_FAILURE);
6 }
7
```

La función `bind` asocia el socket creado en `servidor_fd` a la dirección IP y puerto que hemos configurado en `direccion_servidor`.

Parámetros bind():

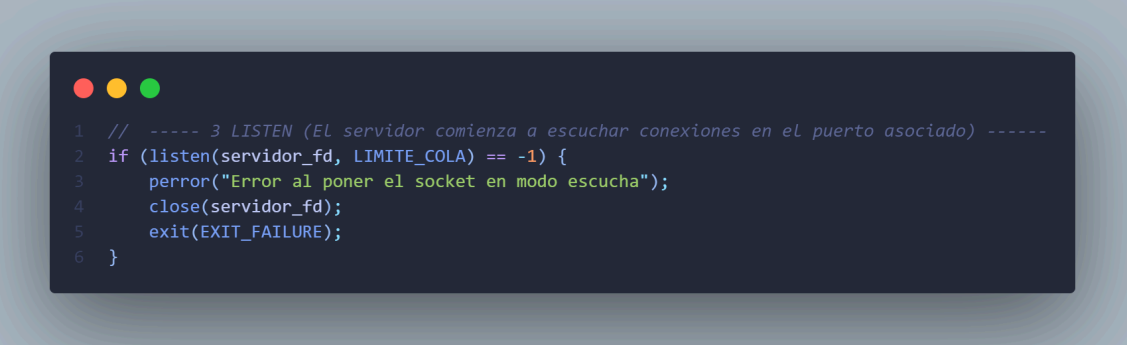
- servidor_fd = Descriptor de socket que se va a asociar.
- (struct sockaddr *)&direccion_servidor = Estructura que contiene los detalles de la dirección IP y el puerto configurado. Se pasa como un puntero genérico para compatibilidad.
- sizeof(direccion_servidor) = Tamaño de la estructura sockaddr_in, que se necesita para que el sistema sepa cuánta memoria leer.

Resumen:

Bind asocia el socket a la IP configurada en sin_addr.s_addr y el puerto configurado en sin_port para que cuando el cliente se conecte, el sistema sepa en qué dirección y puerto está escuchando el servidor.

En caso de que bind falle (-1, puerto en otro programa, dirección IP inválida) se muestra un mensaje de error, se libera el socket creado y se termina el programa.

3 – Poner el socket en modo listen:



```
1 // ----- 3 LISTEN (El servidor comienza a escuchar conexiones en el puerto asociado) -----
2 if (listen(servidor_fd, LIMITE_COLA) == -1) {
3     perror("Error al poner el socket en modo escucha");
4     close(servidor_fd);
5     exit(EXIT_FAILURE);
6 }
```

El servidor se prepara para aceptar conexiones entrantes. Se define el máximo de conexiones en cola que el servidor puede manejar (10).

El servidor ahora está escuchando. Si devuelve 0, es que el servidor está en escucha. Si devuelve -1, ha habido un error. Se mostrará un mensaje de error, se liberará el recurso del socket y se finalizará el programa.

4 – Accept (Aceptar conexiones):

```
1 // Bucle para aceptar y manejar conexiones
2 while (1) {
3     // ----- 4 Accept (El servidor espera una conexión entrante y la acepta) -----
4     cliente_fd = accept(servidor_fd, (struct sockaddr *)&direccion_cliente, &tam_cliente);
5     if (cliente_fd == -1) {
6         perror("Error al aceptar conexión");
7         continue;
8     }
```

Cuando un cliente intenta conectarse, el servidor acepta esa conexión.

- accept() se utiliza para aceptar una nueva conexión.
- servidor fd = descriptor modo escucha (creado por socket() y configurado por bind() y listen()). El socket está preparado para conexiones entrantes.
- (struct sockaddr *)&direccion_cliente = Puntero a estructura sockaddr donde se almacenará información del cliente que se conecta (Dirección IP y puerto).
- &tam_cliente = Puntero al número que contiene tamaño de estructura sockaddr_in (sizeof(direccion_cliente)).

Éxito:

Se devuelve un nuevo descriptor cliente_fd que representa la conexión específica con el cliente.

Error:

Devuelve -1 (cliente cierra conexión antes de completarla). Se omite el resto del bucle y espera a nuevas conexiones.

5 – Crear un proceso hijo(fork):

```
1 // Crear un proceso hijo para manejar la transferencia
2 if (fork() == 0) {
3     close(servidor_fd); // El proceso hijo no necesita el socket del servidor
4     transferirArchivo(cliente_fd); // Transferir el archivo al cliente
5     exit(EXIT_SUCCESS); // Terminar el proceso hijo
6 }
```

Mientras el proceso padre sigue escuchando nuevas conexiones, el proceso hijo maneja la comunicación con el cliente.

El proceso padre cierra la referencia al socket del cliente (ya lo maneja el hijo) en `close(cliente_fd)`.
El proceso hijo llama a `transferirArchivo(cliente_fd)` para manejar la comunicación.

6 – Manejo de la petición (read/write):

```
1 void transferirArchivo(int cliente_fd) {
2     FILE *archivo = fopen(ARCHIVO_ENVIAR, "r"); // Abrir el archivo en modo lectura
3     if (archivo == NULL) {
4         perror("No se pudo abrir el archivo para enviar");
5         close(cliente_fd);
6         return;
7     }
```

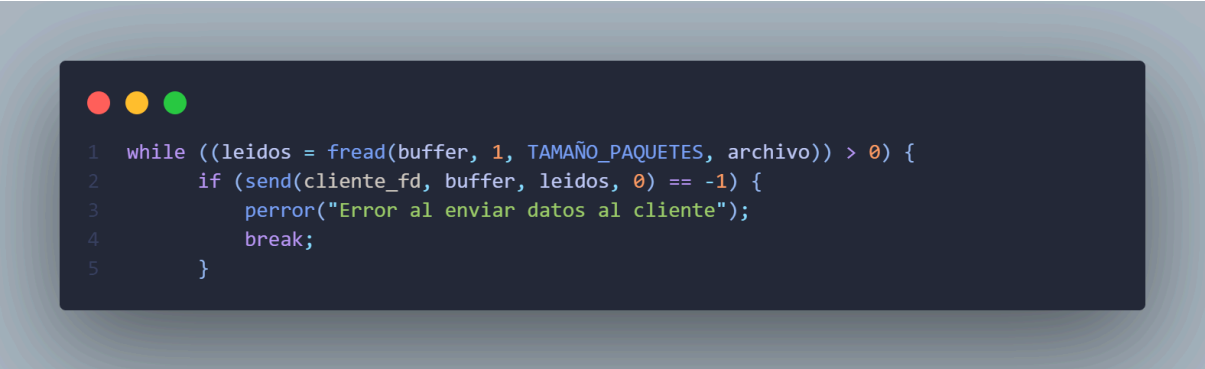
Se abre el archivo en modo lectura (read = r). Si falla al abrir el archivo, se cierra el socket del cliente y no devuelve nada.

```
1 printf("Enviando archivo al cliente...\n");
2 char buffer[TAMAÑO_PAQUETES];
3 size_t leidos;
4
5 // ----- 7 WRITE (El archivo se envía al cliente en bloques de datos) ----- *Leer
  y enviar datos del archivo al cliente
6 while ((leidos = fread(buffer, 1, TAMAÑO_PAQUETES, archivo)) > 0) {
```

Se declara un buffer temporal donde se almacenan bloques de datos del archivo en 1024 bytes, definido en TAMAÑO_PAQUETES.

fread(buffer, 1, TAMAÑO_PAQUETES, archivo) lee hasta 1024 bytes y los guarda en el buffer. Se devuelve el número de bytes leídos.

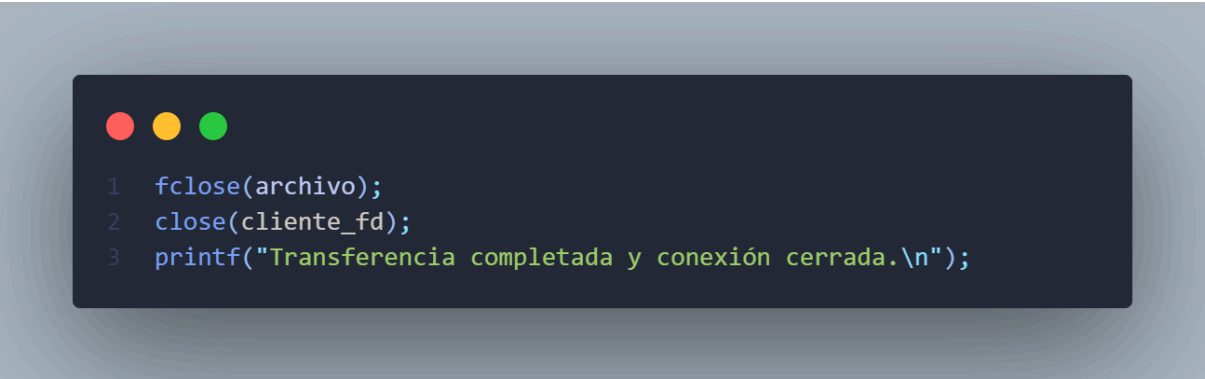
Se empieza una condición while que hace que el bucle siga ejecutando mientras se pueda leer más datos.



```
1 while ((leidos = fread(buffer, 1, TAMAÑO_PAQUETES, archivo)) > 0) {
2     if (send(cliente_fd, buffer, leidos, 0) == -1) {
3         perror("Error al enviar datos al cliente");
4         break;
5     }
```

Se envía el contenido almacenado en el buffer al cliente usando el socket cliente_fd, parámetro de la función.

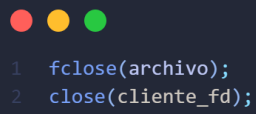
Si send falla, se muestra un error y se rompe el bucle para finalizar el envío.



```
1 fclose(archivo);
2 close(cliente_fd);
3 printf("Transferencia completada y conexión cerrada.\n");
```


Se cierra el archivo después de que se hayan enviado todos los datos, luego se cierra el socket del cliente terminando la conexión para que no se deje el servidor recursos abiertos, y se muestra un mensaje de éxito.

8 – Close (Cerrar el socket):



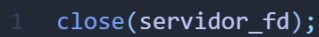
```
1 fclose(archivo);
2 close(cliente_fd);
```

Se cierra el socket del proceso hijo.



```
1 if (fork() == 0) {
2     close(servidor_fd); // El proceso hijo no necesita el socket del servidor
```

Se cierra el socket del proceso padre.

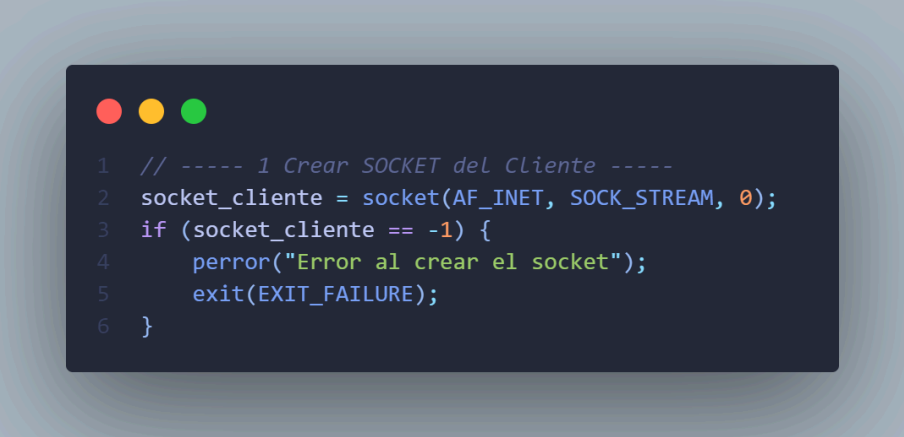


```
1 close(servidor_fd);
```

El socket asociado a la conexión con el cliente se cierra al finalizar el programa.

—Cliente—

1 – Creación del Socket:



```
1 // ----- 1 Crear SOCKET del Cliente -----
2 socket_cliente = socket(AF_INET, SOCK_STREAM, 0);
3 if (socket_cliente == -1) {
4     perror("Error al crear el socket");
5     exit(EXIT_FAILURE);
6 }
```

Se crea un socket TCP/IP (familia de protocolos AF_INET -> IPv4) para la comunicación.

- SOCK_STREAM = Usará TCP (orientado a conexión)
- SOCK_STREAM = Protocolo de transporte TCP (UDP sería SOCK_DGRAM).

* TCP = Protocolo orientado a conexión (asegura que datos lleguen y en orden) y es fiable con control de errores.

- 0 = Selecciona automáticamente protocolo por defecto.

En caso de que el socket no pueda crearse (-1), se muestra un mensaje de error y se termina el programa.

2 – Configuración de la dirección del servidor

```
1 // Configuración de la dirección del servidor
2 servidor_addr.sin_family = AF_INET;
3 servidor_addr.sin_port = htons(PUERTO_SERVIDOR);
4 if (inet_pton(AF_INET, direccionIP, &servidor_addr.sin_addr) <= 0) {
5     perror("Dirección IP inválida o no soportada");
6     close(socket_cliente);
7     exit(EXIT_FAILURE);
8 }
```

Se llena la estructura `servidor_addr` con la información del servidor.

- `sin_family` = Especifica que se usará IPv4.
- `sin_port` = El puerto del servidor (9999) se traduce a `htons` para que pueda entenderlo la red.
- `sin_addr` = Convierte dirección IP a formato binario con `inet_pton`.

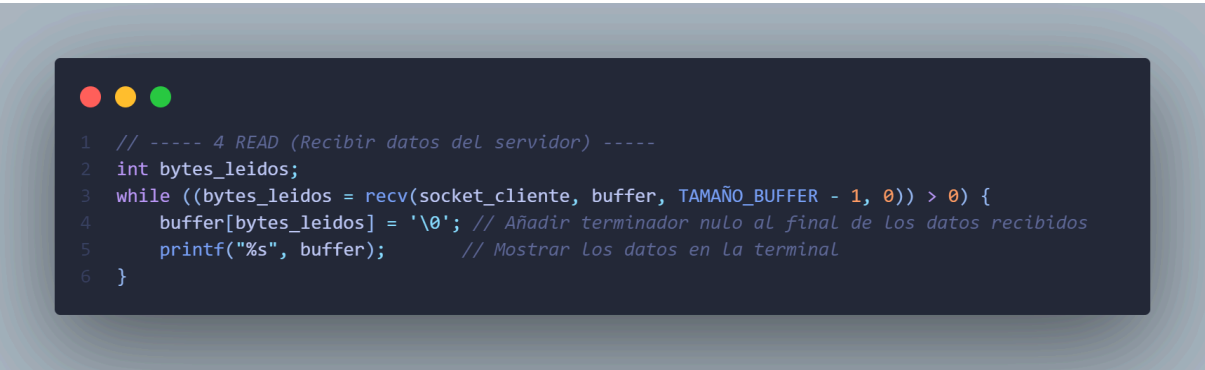
Si la conversión IP falla, se muestra un error y se cierra el socket.

3 – Connect (Conexión al servidor)

```
1 // ----- 3 CONNECT (Cliente conecta con el servicio) -----
2 printf("Conectando al servidor %s en el puerto %d...\n", direccionIP, PUERTO_SERVIDOR);
3 if (connect(socket_cliente, (struct sockaddr *)&servidor_addr, sizeof(servidor_addr)) == -1) {
4     perror("Error al conectar con el servidor");
5     close(socket_cliente);
6     exit(EXIT_FAILURE);
7 }
8 printf("Conexión establecida. Recibiendo datos...\n");
9
```

Se intenta establecer una conexión con el servidor usando la dirección IP y puerto configurado previamente en `servidor_addr`. Si la conexión falla (`connect` devuelve -1), se muestra un mensaje de error y se termina el programa. En caso de que sí que se tenga éxito, se muestra un mensaje diciendo que la conexión ha sido establecida.


4 – Read (Recepción de datos)



```
1 // ----- 4 READ (Recibir datos del servidor) -----
2 int bytes_leidos;
3 while ((bytes_leidos = recv(socket_cliente, buffer, TAMAÑO_BUFFER - 1, 0)) > 0) {
4     buffer[bytes_leidos] = '\0'; // Añadir terminador nulo al final de los datos recibidos
5     printf("%s", buffer);       // Mostrar los datos en la terminal
6 }
```

Se utiliza un bucle para recibir datos enviados por el servidor en bloques. `recv` lee hasta “TAMAÑO_BUFFER - 1”. Esto es porque le agrega un carácter nulo (`\0`) al final del bloque de datos recibido para terminar las cadenas de texto en C.

`printf("%s", buffer)` muestra los datos en la terminal. `%s` indica que se mostrará una cadena de caracteres string que termina con un carácter nulo (`\0`). El buffer contiene los datos recibidos del servidor a través de la función `recv`.



```
1 if (bytes_leidos == -1) {
2     perror("Error al recibir datos");
3 }
```

Aquí se manejan los errores. Si un error ocurre durante la recepción de datos (`recv = -1`), se muestra un mensaje de error.

5 – Close (Cierre de conexión)

```
1     printf("\nTransferencia completa. Cerrando conexión...\n");
2
3     // ----- 5 CLOSE (Cerrar el socket) -----
4     close(socket_cliente);
5 }
```

Una vez los datos han sido recibidos correctamente por el cliente, el servidor cierra la conexión. El cliente muestra un mensaje indicando que la transferencia ha finalizado.

El socket del cliente se cierra con close. De esta manera se liberan recursos del sistema.

6 – Main

```
1 // ----- MAIN -----
2 int main(int argc, char *argv[]) {
3     if (argc != 2) {
4         fprintf(stderr, "Se usa de esta manera: %s <DIRECCIÓN_IP_SERVIDOR>\n", argv[0]);
5         exit(EXIT_FAILURE);
6     }
7
8     recibirArchivo(argv[1]); // Llamar a la función con la IP del servidor
9     return 0;
10 }
```

El main simplemente asegura que a la hora de ejecutar el código se haya pasado el argumento de la dirección IP del servidor. En caso de que no, muestra un mensaje informativo de cómo debe ser y termina el programa.

En caso de que se proporcione la dirección IP, se llama a la función recibirArchivo pasando la dirección IP como argumento.

—Ejecución —

– 1: Servidor

```
PROBLEMS 6 OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
PS C:\Users\mauro\OneDrive\Escritorio\All\IA\Año2\Semestre1> ./run.bat
root@dad9b0b142f7:/workdir# cd S0/Practicas/Practica2
root@dad9b0b142f7:/workdir/S0/Practicas/Practica2# gcc -o Servidor Servidor.c
root@dad9b0b142f7:/workdir/S0/Practicas/Practica2# ./Servidor
Servidor en funcionamiento, esperando conexiones en el puerto 9999...
█
```

Se indica que el servidor a creado un socket asociado al puerto 9999 configurado con el bind, y está esperando a conexiones entrantes con listen.

– 2: Cliente

```
root@dad9b0b142f7:/workdir# cd S0/Practicas/Practica2
root@dad9b0b142f7:/workdir/S0/Practicas/Practica2# gcc -o Cliente Cliente.c
root@dad9b0b142f7:/workdir/S0/Practicas/Practica2# ./Cliente 127.0.0.1
Conectando al servidor 127.0.0.1 en el puerto 9999...
Conexión establecida. Recibiendo datos...
```

El cliente ha creado un socket. Se ha establecido una conexión exitosa con el servidor en la dirección 127.0.0.1 y puerto 9999 utilizando connect.

- 3: Servidor

```
Conexión establecida con 127.0.0.1:45214
Enviando archivo al cliente...
Transferencia completada y conexión cerrada.
```

Indica que efectivamente el servidor ha aceptado la conexión con dirección 127.0.0.1 (localhost) usando accept, y se ha establecido el puerto 45214, que es el puerto asignado automáticamente al cliente por el sistema operativo.

Al completar la transferencia del archivo, el servidor cierra el socket del cliente y muestra “Transferencia completada y conexión cerrada” indicando que ha terminado la tarea para ese cliente.

```

<!doctype html>
<html dir="ltr" lang="es">
  <head>
    <meta charset="utf-8">
    <title>Nueva pestaña</title>
    <style>
      body {
        background: #3C3C3C;
        margin: 0;
      }

      #backgroundImage {
        border: none;
        height: 100%;
        pointer-events: none;
        position: fixed;
        top: 0;
        visibility: hidden;
        width: 100%;
      }

      [show-background-image] #backgroundImage {
        visibility: visible;
      }
    </style>
  </head>
  <body>
    <iframe id="backgroundImage" src=""></iframe>
    <ntp-app></ntp-app>
    <script type="module" src="new_tab_page.js"></script>
    <link rel="stylesheet" href="chrome://resources/css/text_defaults_md.css">
    <link rel="stylesheet" href="chrome://theme/colors.css?sets=ui,chrome">
    <link rel="stylesheet" href="shared_vars.css">
  </body>
</html>
Transferencia completa. Cerrando conexión...
root@dad9b0b142f7:/workdir/SO/Practicas/Practica2# █

```

El cliente comienza a recibir el contenido del archivo enviado por el servidor y lo muestra en la terminal. Este es el contenido del archivo Google.html que el servidor envió. Se recibe en fragmentos y se muestra usando printf.

Una vez terminado de recibir los datos, recv devuelve 0, indicando que el servidor cerró la conexión. Se cierra el socket del cliente.

—Resumen —

Lo que he aprendido en esta práctica es implementar una aplicación cliente-servidor que realiza una transferencia de datos mediante sockets.

El servidor escucha conexiones en el puerto 9999, envía el contenido del archivo Google.html al cliente tras establecer una conexión, y maneja múltiples clientes usando procesos hijos (fork).

El cliente se conecta al servidor específico por una dirección IP (en este caso 127.0.0.1). Recibe el contenido del archivo enviado por el servidor y lo muestra en la terminal.

