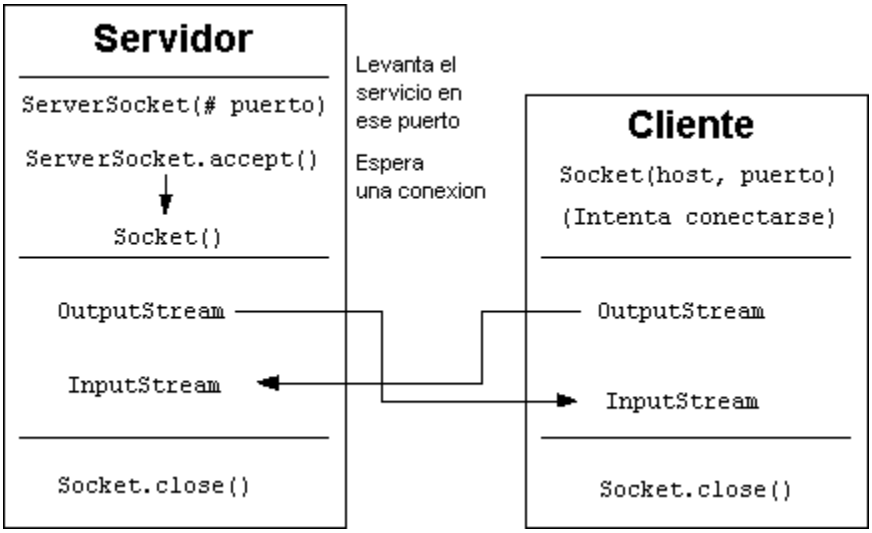
Sockets orientados a conexiones no bloqueantes

Los sockets orientados a conexiones no bloqueantes permiten que un programa continúe ejecutándose sin esperar a que las operaciones de red se completen. Esto es útil en aplicaciones que necesitan mantener una alta responsividad, como servidores web, juegos en línea, o aplicaciones de mensajería.

Un socket no bloqueante se configura para que las operaciones de entrada/salida no detengan el flujo de ejecución del programa si no pueden completarse de inmediato. Esto se logra configurando el socket para que retorne inmediatamente un error (como EWOULDBLOCK o EAGAIN) si la operación no se puede completar.

Se tienen tres características principales:

1. **No bloqueantes:** Las operaciones de red no detienen el flujo del programa.
2. **Asincronía:** Permiten realizar múltiples operaciones simultáneamente.
3. **Eficiencia:** Mejoran la eficiencia en aplicaciones que requieren alta concurrencia.



Como en el ejemplo que se verá más adelante se sigue un flujo general, pero para el caso del manejo de lenguaje C se utilizan ciertos métodos:

Cliente	Servidor
1. Crear un socket. 2. Configurar el socket como no bloqueante. 3. Intentar conectarse al servidor. 4. Continuar con otras operaciones mientras se espera la conexión. 5. Utilizar select() o poll() para verificar si la conexión se ha establecido. 6. Una vez conectado, enviar y recibir datos sin bloquear el flujo del programa.	1. Crear un socket. 2. Configurar el socket como no bloqueante. 3. Ligar el socket a una dirección y puerto. 4. Empezar a escuchar conexiones entrantes. 5. Utilizar select() o poll() para verificar nuevas conexiones. 6. Aceptar nuevas conexiones sin bloquear. 7. Leer y escribir datos de los clientes conectados de manera no bloqueante.

Desarrollo:

Para esta práctica se va a demostrar el funcionamiento de los sockets orientados a conexiones no bloqueantes, por ello, la práctica establece una conexión entre un cliente y un servidor donde el servidor va a aceptar la conexión de diversos clientes sin ninguna restricción.

Para ello se comenzará creando dos archivos:

- Cliente.c
- Servidor.c

Ahora comenzaremos describiendo que es lo que hará el cliente,

Se utilizan sockets, se envía un mensaje y recibe una respuesta. Inicialmente, se definen las variables necesarias: un descriptor de socket (sock), una estructura para la dirección del servidor (serv_addr), un mensaje para enviar al servidor (hello), y un búfer para recibir la respuesta del servidor (buffer). Se procede a crear un socket mediante la función socket(), especificando la familia de direcciones AF_INET para IPv4, el tipo de socket SOCK_STREAM para TCP, y el protocolo 0 para usar el protocolo predeterminado. Si la creación del socket falla, se imprime un mensaje de error y se termina el programa.

La estructura serv_addr se configura con la familia de direcciones AF_INET y el puerto especificado en la macro PORT, convirtiendo el valor a formato de red usando htons(). La dirección IP del servidor se convierte de texto a binario con inet_pton(). Si la conversión falla, se imprime un mensaje de error y se termina el programa. A continuación, se intenta conectar el socket al servidor usando la función connect(), pasando la dirección del servidor. Si la conexión falla, se imprime un mensaje de error y se termina el programa.

Una vez establecida la conexión, se envía el mensaje al servidor utilizando la función send(). Después, se lee la respuesta del servidor con la función read(), almacenando los datos recibidos en el búfer. La respuesta se imprime en la consola. Finalmente, se cierra el socket con la función close() para liberar los recursos asociados. Este flujo permite que el cliente se conecte a un servidor, envíe un mensaje y procese la respuesta de manera eficiente.

Una vez comprendido el flujo que realiza el cliente se procede a agregar el siguiente fragmento de código en el archivo Cliente.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/socket.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char *hello = "Hola desde el cliente";
    char buffer[BUFFER_SIZE] = {0};
```

```
    // Crear socket
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket creation error");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convertir direcciones IPv4 e IPv6 de texto a binario
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        perror("Invalid address/ Address not supported");
        return -1;
    }

    // Conectarse al servidor
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        perror("Connection Failed");
        return -1;
    }

    // Enviar mensaje al servidor
    send(sock, hello, strlen(hello), 0);
    printf("Mensaje enviado\n");

    // Leer respuesta del servidor
    valread = read(sock, buffer, BUFFER_SIZE);
    printf("Respuesta del servidor: %s\n", buffer);

    // Cerrar el socket
    close(sock);

    return 0;
}
```

Se crea un socket del servidor utilizando socket(), especificando la familia de direcciones AF_INET (para IPv4), el tipo de socket SOCK_STREAM (para TCP) y el protocolo (0 para el protocolo predeterminado). Este socket se configura como no bloqueante mediante fcntl(). La dirección del socket se configura con la familia de direcciones AF_INET, la dirección IP INADDR_ANY (que permite aceptar conexiones desde cualquier dirección) y el puerto especificado en la macro PORT. Luego, el socket se une al puerto mediante bind(), y se pone en modo de escucha mediante listen(), permitiendo al servidor aceptar conexiones entrantes.

El bucle principal del servidor se ejecuta indefinidamente, primero limpiando el conjunto de descriptores de archivo con FD_ZERO() y añadiendo el descriptor del servidor a este conjunto con FD_SET(). También se añaden los descriptores de los clientes activos al conjunto. Luego, el servidor espera cualquier actividad en los sockets utilizando select(). Si select() detecta actividad en el socket del servidor, esto indica una nueva conexión entrante, la cual se acepta mediante accept(). El nuevo descriptor del socket del cliente se añade al array client_socket.

Para cada descriptor de cliente activo, el servidor verifica si hay actividad (datos disponibles para leer) utilizando FD_ISSET(). Si un cliente se desconecta (la lectura devuelve 0), se obtiene la información del cliente desconectado con getpeername(), se cierra el socket y se marca como inactivo en el array client_socket. Si se reciben datos, estos se leen en el búfer, se termina la cadena para asegurar la correcta impresión, y se envían de vuelta al cliente usando send().

Una vez comprendido el flujo de lo que realiza el servidor se va a agregar el siguiente fragmento de código al archivo Servidor.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <fcntl.h>
#include <sys/select.h>

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int server_fd, new_socket, max_sd, sd, activity, valread;
    int client_socket[30] = {0};
    int max_clients = 30;
    struct sockaddr_in address;
    char buffer[BUFFER_SIZE];
    fd_set readfds;

    // Crear socket del servidor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Configurar el socket del servidor como no bloqueante
    fcntl(server_fd, F_SETFL, O_NONBLOCK);

    // Configurar el tipo de socket
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Adjuntar el socket al puerto 8080
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind failed");
        close(server_fd);
        exit(EXIT_FAILURE);
    }

    // Escuchar en el socket
    if (listen(server_fd, 3) < 0) {
        perror("listen failed");
        close(server_fd);
        exit(EXIT_FAILURE);
```

```
    }

    printf("Escuchando en el puerto %d \n", PORT);

    // Bucle principal para aceptar y manejar conexiones
    while (1) {
        // Limpiar el conjunto de descriptores de socket
        FD_ZERO(&readfds);

        // Añadir el socket del servidor al conjunto de descriptores
        FD_SET(server_fd, &readfds);
        max_sd = server_fd;

        // Añadir los sockets de cliente al conjunto de descriptores
        for (int i = 0; i < max_clients; i++) {
            sd = client_socket[i];
            if (sd > 0)
                FD_SET(sd, &readfds);
            if (sd > max_sd)
                max_sd = sd;
        }

        // Esperar a que ocurra alguna actividad en uno de los sockets
        activity = select(max_sd + 1, &readfds, NULL, NULL, NULL);
        if (activity < 0) {
            perror("select error");
        }

        // Si hay una actividad en el socket del servidor, es una nueva conexión
        if (FD_ISSET(server_fd, &readfds)) {
            int addrlen = sizeof(address);
            if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen)) < 0) {
                perror("accept failed");
                exit(EXIT_FAILURE);
            }

            printf("Nueva conexión, socket fd es %d, ip es : %s, puerto : %d\n",
new_socket, inet_ntoa(address.sin_addr), ntohs(address.sin_port));

            // Añadir el nuevo socket al array de sockets de cliente
            for (int i = 0; i < max_clients; i++) {
                if (client_socket[i] == 0) {
                    client_socket[i] = new_socket;
                    printf("Añadiendo a la lista de sockets como %d\n", i);
                    break;
                }
            }
        }

        // Manejar IO en otros sockets
        for (int i = 0; i < max_clients; i++) {
            sd = client_socket[i];
            if (FD_ISSET(sd, &readfds)) {
                // Revisar si fue por cierre y leer el mensaje
                if ((valread = read(sd, buffer, BUFFER_SIZE)) == 0) {
```

```
                    // Alguien se desconectó, obtener detalles e imprimir
                    getpeername(sd, (struct sockaddr*)&address,
(socklen_t*)&addrlen);
                    printf("Host desconectado, ip %s, puerto %d\n",
inet_ntoa(address.sin_addr), ntohs(address.sin_port));

                    // Cerrar el socket y marcarlo como 0 en la lista
                    close(sd);
                    client_socket[i] = 0;
                } else {
                    // Poner terminador de cadena en el buffer y enviar mensaje
de vuelta al cliente
                    buffer[valread] = '\0';
                    send(sd, buffer, strlen(buffer), 0);
                }
            }
        }
    }

    return 0;
}
```

Ahora se realiza la ejecución del cliente y servidor, recordemos que la compilación de un archivo en C es:

gcc cliente.c -o cliente y gcc servidor.c -o servidor

La ejecución se ve de la siguiente manera: