1-Explicar qué es un socket y los diferentes tipos de socket.

Concepto de socket

Los programas Linux hacen entradas y salidas escribiendo o leyendo un descriptor de archivo, que es un entero asociado a un archivo abierto. Ese archivo, en este caso, es una conexión de red y no un archivo real. Para las comunicaciones en redes se utiliza la system call ***socket()*** que devuelve un descriptor de comunicaciones para en las llamadas especializadas como read/write/send.

Sockets raw

Se utilizan para acceder directamente a la capa de red o la capa de enlace de datos. Se pueden utilizar para implementar un protocolo de red debajo de la capa de transporte o para monitorear una red. Además, permiten implementar nuevos protocolos en el espacio de usuario.

Sockets de flujo

Definen flujos de comunicación en dos direcciones, confiables y con conexión TCP e IP para el enrutamiento. Si se envían dos mensajes, éstos llegarán al otro extremo ordenados y sin errores.

Sockets de datagrama

Son sockets sin conexión. Si se envían datagramas puede ser que lleguen o no, y si lo hacen, pueden estar desordenados y sin errores. Utilizan IP para el enrutamiento, pero con UDP. Si se quiere asegurar la llegada de los paquetes, se debe implementar un protocolo por encima de UDP.

2-Cuáles son las estructuras necesarias para operar con sockets en el modelo C-S y cómo se hace para ingresar los datos requeridos. Explicar con un ejemplo.

Las estructuras necesarias que se utilizan en los sockets para poder operar en el modelo Cliente-Servidor se encuentran dentro del descriptor de socket, donde una de ellas es [struct sockaddr], la cual mantiene información de direcciones de socket para diversos tipos de sockets, [sa_family] y [sa_data] que contiene una dirección y número de puerto de destino para el socket.

```
    struct sockaddr_in {
        short int          sin_family;  // familia de direcciones, AF_INET
        unsigned short int sin_port;    // Número de puerto
        struct in_addr     sin_addr;    // Dirección de Internet
        unsigned char      sin_zero[8]; // Relleno para preservar el tamaño
original de struct sockaddr
    };
```

Por otro lado, existe una estructura paralela [struct sockaddr_in] para manejar más cómodamente [struct sockaddr], la cual facilita el referirse a los elementos de la dirección de socket, sin_zero rellena todo a ceros para que sea el mismo tamaño que [struct sockaddr], sin_family es el equivalente de [sa_family] y por último sin_port y sin_addr siguen el orden de bytes de la red.

Struct sockaddr_in y struct sockaddr se utilizan para configurar las direcciones y puertos del servidor y del cliente.

3-Explicar modo bloqueante y no bloqueante en sockets y cuáles son los "sockets calls" a los cuales se pueden aplicar estos modos. Explicar las funciones necesarias.

Modo Bloqueante

En modo bloqueante, cuando se realiza una llamada a una función de socket, el programa se bloquea o queda inactivo hasta que la operación en el socket se complete. Las funciones que pueden aplicar estos modos son: accept, recv, recvfrom, listen. Estas "calls" son bloqueantes por defecto y para cambiarlo se llama a la función fcntl().

```
#include <unistd.h>
#include <fcntl.h>
.
.
.
sockfd = socket(PF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

Modo no bloqueante

En modo no bloqueante, las llamadas a funciones de socket retornan inmediatamente, independientemente de si la operación solicitada se ha completado o no. Se requiere un manejo especial para tratar con sockets no bloqueantes, como el uso de bucles y funciones como select() o poll() para esperar eventos en los sockets.

- **Select():** Permite a un programa monitorear múltiples sockets, esperando a que uno o más de ellos estén listos para leer, escribir o tienen excepciones pendientes.
- **Poll():** Similar a select(), pero generalmente más eficiente para manejar grandes números de sockets.

4-**Describir el modelo C-S aplicado a un servidor con Concurrencia Real. Escribir un ejemplo en lenguaje C.**

Un servidor con concurrencia real es aquel capaz de manejar diversas solicitudes simultáneamente, mejorando su eficiencia y capacidad de servicio.
Funciona de la siguiente manera:
_Inicio de la conexión: El cliente inicia una conexión con el servidor enviando una solicitud a través de la red.
_Aceptación de la conexión: El servidor recibe la solicitud de conexión y la acepta, estableciendo así una conexión con el cliente.

_Procesamiento de solicitudes: Una vez establecida la conexión, el servidor espera las solicitudes del cliente. Cuando llega una solicitud, el servidor la procesa según el tipo de servicio solicitado.
_Concurrencia: En un servidor con concurrencia real, el servidor puede manejar múltiples solicitudes simultáneamente. Esto significa que puede atender a varios clientes al mismo tiempo, utilizando técnicas como la multiprogramación, la multitarea o la multiprocesamiento.
_Respuesta: Después de procesar la solicitud, el servidor envía la respuesta correspondiente de vuelta al cliente a través de la conexión establecida.
_Cierre de la conexión: Una vez que se ha completado la transacción, ya sea que se haya proporcionado el recurso solicitado o se haya completado la acción requerida, el servidor cierra la conexión con el cliente.
Esto tiene varias ventajas como la capacidad de distribuir la carga de trabajo entre varios clientes, mejorar la escalabilidad y la disponibilidad del sistema, y permitir un acceso más eficiente a los recursos compartidos. Sin embargo, también tiene desafíos en términos de diseño y gestión de la concurrencia para garantizar un rendimiento óptimo y evitar problemas como las condiciones de carrera y los bloqueos.
Dichos desafíos pueden ser:
Condición de carrera: Ocurre cuando el resultado de la ejecución depende del orden de acceso a los recursos compartidos por parte de múltiples hilos o procesos. Si no se sincroniza adecuadamente el acceso a estos recursos, pueden ocurrir inconsistencias o errores en los datos.

Bloqueos: Los bloqueos pueden ocurrir cuando un recurso está siendo utilizado por un hilo o proceso y otro intenta acceder a él. Si no se manejan correctamente, pueden causar bloqueos de recursos y una degradación del rendimiento.

Deadlocks: Un deadlock (o interbloqueo) ocurre cuando dos o más procesos se bloquean entre sí porque cada uno espera un recurso que está siendo utilizado por otro proceso. Esto puede provocar una situación en la que ninguno de los procesos pueda continuar.

Consumo excesivo de recursos: La concurrencia puede aumentar el consumo de recursos del sistema, como la memoria y la capacidad de procesamiento, especialmente si se maneja de manera ineficiente. Si no se controla adecuadamente, esto puede llevar a problemas de rendimiento y a una menor capacidad de escalabilidad del servidor.

Gestión de hilos y procesos: La creación, gestión y sincronización de hilos o procesos en un entorno concurrente puede ser compleja y propensa a errores. Se requiere un

cuidadoso diseño y gestión para garantizar que los recursos se utilicen de manera eficiente y que se minimicen los problemas de concurrencia.

Debugging y testing: Depurar y probar aplicaciones concurrentes puede ser más difícil que las aplicaciones secuenciales debido a la naturaleza impredecible de la concurrencia. Los errores pueden ser difíciles de reproducir y diagnosticar, lo que hace que sea crucial implementar técnicas de depuración y pruebas específicas para la concurrencia.

Para abordarlos, se utilizan diversas técnicas y herramientas, como el uso de mecanismos de sincronización (como semáforos, mutex, monitores), la implementación de protocolos de exclusión mutua, el diseño de algoritmos de planificación eficientes y la adopción de prácticas de programación concurrente seguras.

Codigo del servidor:

```c
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h> // read(), write(), close()
#include <arpa/inet.h> // Para inet_ntop()
#define MAX 80
#define PORT 7800
#define SA struct sockaddr

// Función para el chat entre el cliente y el servidor
void func(int connfd, int valor)
{
    char buffer[MAX];
    int n;
    // Loop para el chat
    for (;;)
    {
        bzero(buffer, MAX);

        // Leo el mensaje del cliente y lo copio en un buffer
        read(connfd, buffer, sizeof(buffer));
        // Muestro el buffer con los datos del cliente
        printf("Del cliente[%d]: %s\t: ",valor, buffer);
        bzero(buffer, MAX);
        n = 0;
```

```
        // Copio el mensaje del servidor en el buffer
        while ((buffer[n++] = getchar()) != '\n')
            ;

        // y envío el buffer al cliente
        write(connfd, buffer, sizeof(buffer));

        // si el mensaje dice "SALIR" salgo y cierro conexión
        if (strncmp("SALIR", buffer, 4) == 0)
        {
            printf("Cliente desconectado...\n");
            break;
        }
    }
}

// Función principal
int main()
{
    int sockfd, connfd, len;
    pid_t childpid;
    struct sockaddr_in servaddr, cli;

    // socket create and verification
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1)
    {
        printf("Falla la creación del socket...\n");
        exit(0);
    }
    else
        printf("Socket creado...\n");
    bzero(&servaddr, sizeof(servaddr));

    // asigno IP, PORT
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORT);

    // Bind del nuevo socket a la dirección IP y lo verifico
    if ((bind(sockfd, (SA *)&servaddr, sizeof(servaddr))) != 0)
    {
        printf("Falla socket bind ...\n");
        exit(0);
    }
    else
        printf("Se hace el socket bind ..\n");
```

```c
    char server_ip[16]; // Almacena la dirección IP del servidor
    inet_ntop(AF_INET, &(servaddr.sin_addr), server_ip, 16 );
    printf("Dirección IP del servidor: %s\n", server_ip);
    printf("Puerto del servidor: %d\n", PORT);

    // El servidor está en modo escucha (pasivo) y lo verifico
    if ((listen(sockfd, 5)) != 0)
    {
        printf("Falla el listen ...\n");
        exit(0);
    }
    else
        printf("Servidor en modo escucha ...\n");
    len = sizeof(cli);
    int valor = 0;

    // Loop para aceptar conexiones de clientes
    for (;;)
    {
        valor++;
        // Acepto los paquetes del cliente y verifico
        connfd = accept(sockfd, (SA *)&cli, &len);
        if (connfd < 0)
        {
            printf("Falla al aceptar datos en el servidor...\n");
            exit(0);
        }
        else
            printf("El servidor acepta al cliente[%d] ...\n", valor);

        // Creo un nuevo proceso hijo para manejar la conexión del cliente
        if ((childpid = fork()) == 0)
        {
            // Estamos en el proceso hijo
            close(sockfd); // El hijo no necesita el socket de escucha

            // Procesamos la conexión con el cliente
            func(connfd,valor);

            // Cerramos la conexión del cliente
            close(connfd);

            // Salimos del proceso hijo
            exit(0);
        }

        // Cerramos el socket de conexión en el proceso padre
        close(connfd);
```

```
    }

    // Cerramos el socket de escucha
    close(sockfd);

    return 0;
}
```

Codigo del cliente:

```
#include <arpa/inet.h> // inet_addr()
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h> // bzero()
#include <sys/socket.h>
#include <unistd.h> // read(), write(), close()
#define MAX 80
#define PORT 6800
#define SA struct sockaddr

void func(int sockfd, struct sockaddr_in cli)
{
    char buff[MAX];
    int n;
    char *client_ip = inet_ntoa(cli.sin_addr);
    int client_port = ntohs(cli.sin_port);
    printf("Cliente conectado desde: %s:%d\n", client_ip, client_port);
    for (;;) {
        bzero(buff, sizeof(buff));
        printf("Ingrese texto : ");
        n = 0;
        while ((buff[n++] = getchar()) != '\n')
            ;
        write(sockfd, buff, sizeof(buff));
        bzero(buff, sizeof(buff));
        read(sockfd, buff, sizeof(buff));
        printf("Servidor : %s", buff);
        if ((strncmp(buff, "exit", 4)) == 0) {
            printf("Cliente cierra conexión...\n");
            break;
        }
    }
}

int main()
{
    int sockfd, connfd;
```

```
    struct sockaddr_in servaddr, cli;

    // socket: creo socket y lo verifico
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        printf("Falla la creación del socket...\n");
        exit(0);
    } else
        printf("Socket creado ..\n");
    bzero(&servaddr, sizeof(servaddr));

    // asigno IP, PORT
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(PORT);

    // Conecto los sockets entre cliente y servidor
    if (connect(sockfd, (SA *)&servaddr, sizeof(servaddr)) != 0) {
        printf("Falla de conexión con servidor...\n");
        exit(0);
    } else
        printf("Conectado al servidor..\n");

    // Obteniendo información del cliente
    struct sockaddr_in client_addr;
    socklen_t client_len = sizeof(client_addr);
    getpeername(sockfd, (struct sockaddr *)&client_addr, &client_len);

    // Función para el chat
    func(sockfd, client_addr);

    // Cierro el socket
    close(sockfd);
}
```

5-Cómo se cierra una conexión C-S ?. Métodos que eviten la pérdida de informacion.

En el momento que se quiera cerrar la conexión desde algunos de los dos extremos hay dos modos de cerrar la conexión:

1- De forma abrupta donde se utiliza el comando close(sockfd) y el socket no recibe ningún read o write, en caso de que un cliente intente utilizar un read o write va a recibir un mensaje de error.

2- De forma organizada, en este caso se utiliza el comando shutdown(int sockfd, int how) donde how es la forma que especifica como se quiere cerrar la sesión, hay 3 formas de cerrar la conexión:

- 0: No se permite recibir más datos
- 1: No se permite enviar más datos
- 2: No se permite enviar ni recibir más datos.

A su vez cuando termina de ejecutarse el shutdown se debe utilizar un close ya que es la única forma de liberar el socket.

6-**Probar el código del chat entre cliente y servidor. Cambiar tipo de socket y volver a probar.**

Para esta parte cuando creamos el socket cambiamos su tipo de SOCK_STREAM a SOCK_DGRAM esta diferencia es cuando un cliente se conecta por flujo o por datagrama y en cualquiera de los dos casos el servidor anda igual.