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

Un socket es una interfaz de comunicación, permite a dos dispositivos o programas diferentes en una red comunicarse entre sí. Proporciona un punto final en la comunicación entre estos con el fin de enviar y recibir datos a través de este socket.
Hay cuatro tipos de socket. Primero está el Socket de datagrama o más conocido como UDP socket, ofrece una comunicación no orientada a la conexión, pero no tiene la garantía de la entrega entre los sockets (pueden llegar en un orden diferente al que fueron enviados o parte del paquete perderse en la comunicación).
Después, es el Packet socket en donde se lo utiliza para capturar y enviar paquetes individuales en una red, concede un acceso de bajo nivel a la capa de red y se lo utiliza para apps de monitoreo de red y análisis de tráfico.
Luego, está el socket de flujo o TCP socket, que a diferencia del datagramsocket suministra una conexión orientada a la conexión y confiable entre dos hosts en la red. Se utiliza para datos de flujo continuo y garantiza la llegada de los paquetes en el mismo orden enviado.
Por último, está el raw socket, que es el socket bruto en donde ofrece acceso de bajo nivel a la capa de red y permite enviar y recibir paquetes sin procesar

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 son específicas dependiendo de si estamos del lado del servidor o el cliente. Para el servidor son:La dirección del servidor[struct sockaddr] la cual contiene diversos tipos de sockets [sa_family] Especifica la familia de dirección, que puede ser AF_INET para IPv4, AF_INET6 para IPv6, u otras constantes que representan diferentes familias de direcciones de socket. [sa_data] Almacena la dirección en formato binario. La longitud y el formato de esta área dependen de la familia de direcciones especificada en sa_family,un socket de escucha[socket()] el cual espera nuevas conexiones de clientes, después el [sockaddr_in],en el cual simplifica la administración de direcciones, el campo [sin_zero] crea compatibilidad con [struct sockaddr] para definir tamaño de datos y rellenar con ceros utilizando [memset()]. Por último, está el in_addr en donde se almacena la dirección IP en formato red y sirve para inicializar y manipular [sin_family],[sin_port] y [sin_addr]
Por otro lado, el cliente necesita de la dirección del servidor, un socket del cliente y las estructuras para enviar y recibir datos del servidor.

```
struct sockaddr {
    unsigned short    sa_family;    // Familia de dirección: AF_INET, AF_INET6, etc.
    char              sa_data[14];  // Almacenamiento de la dirección (dependiente de la familia)
};

struct sockaddr_in {
    short          sin_family;   // Familia de dirección (siempre AF_INET)
    unsigned short   sin_port;     // Número de puerto en formato de red
    struct in_addr   sin_addr;     // Dirección IP en formato de red
    char           sin_zero[8];  // Relleno para que tenga el mismo tamaño que sockaddr
};
struct in_addr {
```

```
    uint32_t s_addr;  // Dirección IP en formato de red
};
```

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.

El modo bloqueante y no bloqueante en sockets son llamadas a funciones del comportamiento de los sockets en términos de bloqueo del proceso.
En el modo bloqueante las llamadas a las funciones de socket hacen que el proceso que las realiza se bloquee hasta que la operación correspondiente se complete.
Por ejemplo, si un proceso llama a la función recv() para recibir datos de un socket en modo bloqueante y no hay datos disponibles para recibir, el proceso se bloqueará y esperará hasta que se reciban datos o se produzca un error.
Por el otro lado,en el modo no bloqueante, las llamadas a las funciones de socket no bloquean el proceso, incluso si la operación solicitada no se puede completar de inmediato. En lugar de bloquearse, las funciones de socket en modo no bloqueante devuelven un valor especial que indica que la operación no se pudo completar en ese momento.
Esto permite que el proceso continúe ejecutándose y realice otras tareas mientras espera que ocurra algún evento relacionado con el socket, como la disponibilidad de datos para recibir o la capacidad de enviar datos.

Los socket calls que se pueden aplicar en estos modos son:

- socket() - Crea un nuevo socket.
- bind() - Asocia una dirección IP y un número de puerto a un socket.
- listen() - Cambia el estado de un socket en estado de escucha para aceptar conexiones entrantes.
- accept() - Acepta una conexión entrante en un socket de escucha.
- connect() - Establece una conexión a un socket remoto.
- send() - Envía datos a través de un socket.
- recv() - Recibe datos desde un socket.

Las funciones necesarias para el modo no bloqueante y bloqueante aparte de las funciones de socket call son las funciones para monitorear el estado del socket y gestionar la operación no bloqueante estas son:**fcntl()**,utilizada para establecer o cambiar el modo de bloqueo de un descriptor de archivo, incluidos los sockets.**select()** , en donde permite al proceso monitorear múltiples descriptores de archivo, incluidos los sockets, y determinar cuáles están listos para lectura, escritura o excepción. **poll()** el cual ofrece una interfaz similar a select() pero con mayor flexibilidad y eficiencia en algunas situaciones.Por último, **epoll()** en donde proporciona una interfaz eficiente para monitorear múltiples descriptores de archivo en sistemas Linux.

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

El enfoque Cliente-Servidor con concurrencia real permite al servidor gestionar múltiples clientes simultáneamente. Esta capacidad se logra mediante el uso de técnicas como la creación de subprocesos o procesos hijos para manejar las solicitudes individuales de cada cliente. Esta arquitectura permite al servidor atender a varios clientes al mismo tiempo sin

esperar a que una solicitud previa se complete, lo que mejora significativamente la eficiencia y la capacidad de respuesta del sistema en entornos con alta carga de trabajo.

Modelo del cliente

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/wait.h>

#define MAX 80
#define PORT 8080
#define SA struct sockaddr

void func(int sockfd,pid_t childpid){
    // Loop para el chat con el servidor
    char buff[MAX];
    int num;

    for (;;) {
        printf("Ingrese un número (o 'exit' para salir): ");
        bzero(buff, MAX);
        fgets(buff, sizeof(buff), stdin);

        // Enviar el mensaje al servidor
        write(sockfd, buff, strlen(buff));

        // Si el mensaje es "exit", salir del bucle
        if (strncmp("exit", buff, 4) == 0) {
            printf("Cliente cierra conexión...\n");
            break;
        }

        // Leer la respuesta del servidor
        bzero(buff, MAX);
        read(sockfd, buff, sizeof(buff));

        // Imprimir el resultado del factorial
        sscanf(buff, "%d", &num);
```

```
        printf("Factorial de %s es : %d\n", buff, num);

        // Fork para permitir múltiples clientes simultáneos
        if ((childpid = fork()) == 0) { // Proceso hijo
            close(sockfd); // Cerrar socket en el proceso hijo
            break; // Salir del bucle en el proceso hijo
        }
    }
}

int main() {
    int sockfd, connfd;
    struct sockaddr_in servaddr, cli;
    pid_t childpid; // Variable para el identificador del proceso hijo

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

    // Conectar el socket al servidor
    if (connect(sockfd, (SA*)&servaddr, sizeof(servaddr)) != 0) {
        printf("Falla al conectar al servidor...\n");
        exit(0);
    }
    printf("Conectado al servidor..\n");

    func(sockfd,childpid);

    // Cerrar el socket al salir
    close(sockfd);

    // Esperar a que todos los procesos hijos terminen
    while (wait(NULL) > 0);
```

```
    return 0;
}
```

Modelo del servidor

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/wait.h>

#define MAX 80
#define PORT 8080
#define SA struct sockaddr

int factorial(int n) {
    if (n == 0 || n == 1) return 1;
    else return n * factorial(n - 1);
}

int main() {
    int sockfd, connfd, len;
    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);
    }
    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);

    // asigno IP, PORT
    if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
        printf("Falla al hacer el socket bind..\n");
```

```
        exit(0);
    }
    printf("Se hace el socket bind..\n");

    // Servidor en modo escucha
    if ((listen(sockfd, 5)) != 0) {
        printf("Falla al poner el servidor en modo escucha...\n");
        exit(0);
    }
    printf("Servidor en modo escucha ...\n");
    len = sizeof(cli);

    // Loop para aceptar múltiples clientes
    for (;;) {
        // Aceptar el cliente
        connfd = accept(sockfd, (SA*)&cli, &len);
        if (connfd < 0) {
            printf("El servidor no pudo aceptar al cliente...\n");
            exit(0);
        }
        printf("El servidor acepta al cliente ...\n");

        // Fork para manejar la interacción con el cliente
        pid_t pid = fork();
        if (pid == 0) { // Proceso hijo
            close(sockfd); // Cerrar socket en el proceso hijo

            // Loop para manejar la interacción con el cliente
            char buff[MAX];
            int num;

            for (;;) {
                bzero(buff, MAX);

                // Leer el mensaje del cliente
                read(connfd, buff, sizeof(buff));
                printf("Mensaje recibido del cliente: %s\n", buff);

                // Si el mensaje es "exit", salir del bucle
                if (strncmp("exit", buff, 4) == 0) {
                    printf("Cliente solicita salir...\n");
                    break;
                }
```

```
                // Calcular el factorial del número recibido
                sscanf(buff, "%d", &num);
                int result = factorial(num);

                // Enviar el resultado al cliente
                bzero(buff, MAX);
                sprintf(buff, "%d", result);
                write(connfd, buff, sizeof(buff));
                printf("Factorial calculado por el servidor: %d\n",
result);
            }

            // Cerrar la conexión al cliente
            close(connfd);
            exit(0);
        } else if (pid < 0) {
            printf("Falla en fork() al manejar cliente...\n");
            exit(0);
        }

        // Cerrar la conexión al cliente en el proceso padre
        close(connfd);
    }

    // Cerrar el socket al salir
    close(sockfd);
    return 0;
}
```

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

Se cierra siguiendo un procedimiento específico dependiendo del protocolo de comunicación utilizado y la implementación particular. Algunos métodos que evitan la pérdida de información son:
A- Que el cliente o servidor **envíe un mensaje de cierre** antes de cerrar la conexión para notificar que esta será finalizada. Esto permite que realicen cualquier acción necesaria, como por ejemplo realizar una limpieza de recursos y enviar los últimos datos.
B- **El protocolo de cierre explícito**, en este por ejemplo, el protocolo TCP cuando desea cerrar la conexión envía un paquete FIN al otro lado, que responde con un

ACK y luego envía su propio paquete ACK. Después de recibir esta confirmación, el primer lado envía una última para confirmar el cierre.
C- **Temporizador de inactividad**, si un cliente o servidor deja de recibir o enviar datos fuente un cierto tiempo, se puede cerrar la conexión automáticamente para evitar la inactividad, todo esto mediante un temporizador.
D- **Manejo de errores**, en este caso, si se produce un error durante la comunicación, que puede ser por ejemplo una conexión interrumpida, se puede cerrar la misma de manera inmediata para evitar la pérdida de datos adicionales. Es importante, con este método, manejar correctamente los errores como también notificar al otro extremo sobre la terminación de la conexión.

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

Realizamos el cambio de socket de SOCK_STREAM A SOCK_DGRAM, en el cual el cliente se conecta por flujo o por datagrama. Se observó que los resultados son similares.
Resultado de chat entre cliente y servidor con sockets de datagramas: