PROTOCOLOS DE INTERNET

Trabajo Práctico N.º1 :Modelo Cliente-Servidor

1 CUATRIMESTRE DE 2023

Trabajo Práctico Nº 1

Grupo: 17

Profesor: Javier Ouret

Integrantes:

Nº	Apellido y Nombre	Carrera	Legajo	E mail
1	Cortina, Tomás	Ing. Informática	152154763	tomas.cortana@uca.edu.ar
2	Denti, Mateo	Ing. Informática	152151542	mateo.denti@outlook.com.ar
3	Fantino, Thiago	Ing. Informática	152151054	thiagofantino@uca.edu.ar
4	Soriano, Máximo	Ing. Informática	152154633	maximo.soriano170103@gmail.com

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

Los sockets son puntos de conexión fundamentales que permiten la comunicación bidireccional entre programas en dispositivos diferentes a través de una red. Cuando dos programas desean comunicarse, uno actúa como servidor, que espera conexiones entrantes, mientras que el otro actúa como cliente, iniciando la conexión. Un socket está asociado con un protocolo de comunicación específico, una dirección IP y un número de puerto, lo que lo hace único en la red.

Existen tres tipos principales de sockets:

Sockets de flujo (stream sockets): Utilizados para comunicación fiable y orientada a la conexión, como TCP. Proporcionan un flujo de datos bidireccional y confiable entre el cliente y el servidor, ideal para aplicaciones donde la integridad y el orden de los datos son críticos, como transferencia de archivos o navegadores web.

Sockets de datagrama (datagram sockets): Utilizados para comunicación no fiable y sin conexión, como UDP. Transmiten datos en forma de datagramas individuales, que pueden llegar desordenados o incluso perderse. Son adecuados para aplicaciones donde la velocidad y la simplicidad son más importantes que la fiabilidad, como videojuegos en línea o servicios de tiempo.

Sockets puros (raw sockets): Permiten el acceso directo a la capa de red sin procesar, lo que facilita la creación y manipulación de paquetes de red a nivel de transporte y de red. Se utilizan principalmente para tareas de bajo nivel, como análisis de tráfico o desarrollo de herramientas de diagnóstico.

Además, los sockets pueden funcionar con diferentes protocolos de red, como TCP o UDP, dependiendo de las necesidades de la aplicación. En resumen, los sockets son canales esenciales que permiten a los procesos intercambiar datos y mensajes, ya sea localmente o a través de redes, utilizando diferentes tipos y protocolos según los requisitos específicos de la comunicación.

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.

En el modelo Cliente-Servidor, se utilizan varias estructuras y funciones para establecer y gestionar la comunicación entre el cliente y el servidor. Las estructuras principales necesarias son:

- struct sockaddr: Esta estructura se utiliza para representar direcciones de socket sin importar el protocolo de red específico, es decir, mantiene informacion de direcciones de socket para diversos tipos. Aquí está su definición:

```
struct sockaddr {
   unsigned short sa_family;       // Familia de direcciones
   char sa_data[14];               // Bytes de la dirección del protocolo
};
```

[sa_family] admite varios valores, pero será AF_INET en general
[sa_data] contiene una dirección y número de puerto de destino para el socket.
No es adecuado empaquetar a mano una dirección y un número de puerto dentro de sa_data .

- struct sockaddr_in: Esta estructura se utiliza para almacenar información sobre la dirección IP y el número de puerto del servidor al que el cliente desea conectarse, así como para almacenar la dirección y el número de puerto del cliente en el servidor. Existe para manejar struct sockaddr más cómodamente. “in” es por “Internet”. Aquí está su definición:

```
struct sockaddr_in {
    short int sin_family;           // Familia de direcciones (usualmente 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
};
```

Esta estructura hace más sencillo referirse a los elementos de la dirección de socket.
sin_zero (que se incluye para que la nueva estructura tenga el mismo tamaño struct sockaddr) debe rellenarse todo a ceros usando la función memset().
un puntero a struct sockaddr_in puede forzarse [cast] a un puntero a struct sockaddr y viceversa.

Para inicializar estas estructuras y operar con sockets, se deben asignar valores adecuados a sus campos. Por ejemplo, al crear un socket del lado del servidor, se inicializa una estructura struct sockaddr_in con la dirección IP y el número de puerto en los que se espera que escuche el servidor. Luego, se llama a la función bind() para asociar esta dirección con el socket.

Para crear un socket del lado del cliente, se inicializa una estructura struct sockaddr_in con la dirección IP y el número de puerto del servidor al que se desea conectar. Luego, se llama a la función connect() para establecer una conexión con el servidor.

A continuación, se presenta un ejemplo de cómo se inicializan estas estructuras para crear un socket del lado del servidor:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT 8080

int main(int argc, char const *argv[]) {
```

```
    int server_fd, new_socket, valread;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
    char *message = "¡Hola, cliente! Recibi tu mensaje.";

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

    // Asignar dirección y puerto al socket
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);
    if (bind(server_fd, (struct sockaddr *) &address, sizeof(address)) < 0) {
        perror("Error al asignar dirección y puerto al socket");
        exit(EXIT_FAILURE);
    }

    // Escuchar conexiones entrantes
    if (listen(server_fd, 3) < 0) {
        perror("Error al escuchar conexiones entrantes");
        exit(EXIT_FAILURE);
    }

    printf("Servidor listo para aceptar conexiones\n");

    // ...
}
```

Este ejemplo crea un socket del lado del servidor que escucha en el puerto 8080. La estructura address de tipo struct sockaddr_in se utiliza para asignar la dirección IP y el número de puerto al socket mediante la función bind(). Luego, el servidor está listo para aceptar conexiones entrantes.

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 programación de sockets se refieren a cómo las llamadas de sistema relacionadas con sockets se comportan en términos de bloqueo del proceso que las llama.

En el modo bloqueante, cuando un proceso realiza una llamada a una función de socket, el proceso se bloquea hasta que la operación se complete. Durante este tiempo de bloqueo, el proceso no puede realizar otras operaciones, lo que significa que está esperando activamente a que se complete la operación de socket. Por ejemplo, al llamar a funciones

como recv() o send() en modo bloqueante, el proceso se bloqueará hasta que se reciban o envíen los datos.

En contraste, en el modo no bloqueante, cuando un proceso realiza una llamada a una función de socket, el proceso no se bloquea. La función de socket devuelve inmediatamente, incluso si la operación no se puede completar de inmediato. Esto permite que el proceso continúe ejecutándose y realice otras operaciones mientras espera que se complete la operación de socket. Por ejemplo, al usar funciones como recv() o send() en modo no bloqueante, el proceso puede continuar ejecutándose incluso si no se pueden recibir o enviar datos de inmediato.

Las llamadas de sistema y funciones asociadas en las que se pueden aplicar estos modos son diversas, algunas de las más comunes son:

- recv() y recvfrom(): Para recibir datos de un socket.
- send() y sendto(): Para enviar datos a un socket.
- accept(): Para aceptar una conexión entrante en un socket de escucha.
- connect(): Para conectarse a un socket remoto.
- select(), poll(), epoll(), etc.: Para esperar eventos en múltiples sockets de manera eficiente.

Para activar el modo no bloqueante en un socket, se utiliza la función fcntl(). Esta función se utiliza para establecer la opción en el socket que indica si se debe usar el modo bloqueante o no bloqueante. Aquí hay un ejemplo de cómo se activa el modo no bloqueante en un socket en C:

```
#include <fcntl.h>
#include <sys/socket.h>

int sockfd = socket(AF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

En este ejemplo, después de crear el socket con socket(), se utiliza fcntl() para establecer la opción O_NONBLOCK, lo que hace que el socket sea no bloqueante. De esta manera, las llamadas subsiguientes a funciones de socket, como recv() o send(), se comportarán en modo no bloqueante.

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

El modelo Cliente-Servidor (C-S) es una arquitectura común en sistemas distribuidos donde un servidor ofrece servicios a múltiples clientes. En el contexto de un servidor con concurrencia real, el servidor está diseñado para manejar múltiples conexiones de clientes de manera simultánea, permitiendo que atienda las solicitudes de varios clientes sin bloquear su flujo de trabajo.

En este modelo:

1. Servidor:

- El servidor se ejecuta como un proceso independiente y espera conexiones de clientes.
- Cuando un cliente se conecta, el servidor crea un nuevo proceso hijo (en el ejemplo) para manejar la comunicación con ese cliente.
- El servidor puede manejar múltiples conexiones de clientes de manera simultánea utilizando procesos hijos o algún otro mecanismo de concurrencia, como hilos de ejecución.
- Cada proceso hijo del servidor está dedicado a interactuar con un cliente específico, lo que permite que el servidor atienda a varios clientes de forma concurrente.

2. Cliente:
- El cliente se conecta al servidor utilizando su dirección IP y número de puerto.
- Una vez establecida la conexión, el cliente puede enviar solicitudes al servidor y recibir respuestas.

3. Comunicación:
- La comunicación entre el cliente y el servidor se realiza a través de sockets, utilizando un protocolo de red como TCP.
- El servidor escucha en un puerto específico y espera conexiones entrantes de los clientes.
- Cuando un cliente se conecta, el servidor acepta la conexión y crea un nuevo proceso hijo para manejar la comunicación con ese cliente.
- El servidor procesa las solicitudes del cliente, realiza las operaciones necesarias y envía las respuestas de vuelta al cliente.
- Mientras el servidor está ocupado atendiendo a un cliente, puede seguir aceptando conexiones de otros clientes y manejar sus solicitudes simultáneamente.

Este es un ejemplo en lenguaje C de un servidor que maneja múltiples conexiones de clientes utilizando procesos hijos:

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

#define PORT 8080

void proceso_hijo(int conn_sock) {
    char buffer[1024];
    ssize_t bytes_recv;

    printf("Conexión establecida con el cliente\n");

    // Enviamos un mensaje al cliente
    send(conn_sock, "Servidor: Conectado con cliente\n", 32, 0);
```

```
    while (1) {
        // Recibimos datos del cliente
        bytes_recv = recv(conn_sock, buffer, sizeof(buffer), 0);
        if (bytes_recv <= 0) {
            printf("Cliente desconectado\n");
            break;
        }

        printf("Recibido: %.*s\n", (int)bytes_recv, buffer);

        // Enviamos los datos de vuelta al cliente
        send(conn_sock, buffer, bytes_recv, 0);
    }

    // Cerramos el socket de conexión
    close(conn_sock);
}

int main() {
    int sockfd, conn_sock;
    struct sockaddr_in serv_addr, cli_addr;
    socklen_t cli_len = sizeof(cli_addr);

    // Creamos el socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("Error al abrir el socket");
        exit(EXIT_FAILURE);
    }

    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(PORT);

    // Ligamos el socket a la dirección y puerto
    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
        perror("Error al enlazar el socket");
        exit(EXIT_FAILURE);
    }

    // Ponemos el socket en modo de escucha
    if (listen(sockfd, 5) < 0) {
        perror("Error al poner el socket en modo escucha");
        exit(EXIT_FAILURE);
    }

    printf("Socket en modo escucha - pasivo\n");
```

```
    while (1) {
        // Aceptamos una conexión entrante
        conn_sock = accept(sockfd, (struct sockaddr *) &cli_addr, &cli_len);
        if (conn_sock < 0) {
            perror("Error al aceptar la conexión");
            exit(EXIT_FAILURE);
        }

        // Creamos un proceso hijo para manejar la conexión
        pid_t pid = fork();
        if (pid < 0) {
            perror("Error al crear el proceso hijo");
            exit(EXIT_FAILURE);
        } else if (pid == 0) {
            // Estamos en el proceso hijo
            close(sockfd); // Cerramos el socket del servidor en el proceso hijo
            proceso_hijo(conn_sock); // Manejamos la conexión con el cliente
            exit(EXIT_SUCCESS);
        } else {
            // Estamos en el proceso padre
            close(conn_sock); // Cerramos el socket de conexión en el proceso padre
        }
    }

    // Cerramos el socket del servidor
    close(sockfd);

    return 0;
}
```

Este código crea un servidor TCP que escucha en un puerto específico y maneja las conexiones entrantes de los clientes en procesos hijos. Cada proceso hijo se encarga de la comunicación con un cliente específico, enviando un mensaje de saludo al cliente cuando se conecta.

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

Cerrar una conexión Cliente-Servidor (C-S) de manera adecuada es esencial para asegurar la integridad de la comunicación y evitar la pérdida de información. Estos son los métodos comunes para cerrar una conexión C-S y los métodos que evitan la pérdida de información:

Métodos de cierre de conexión C-S:

1. Cierre normal:
   - Ambos extremos envían un mensaje especial para indicar que desean cerrar la conexión.

- Después de recibir este mensaje, el otro extremo también cierra su socket de manera ordenada.
- Este método garantiza que todos los datos pendientes se envíen correctamente antes de cerrar la conexión.

2. Cierre forzado:
- Una parte cierra su socket sin enviar un mensaje de cierre al otro extremo.
- El otro extremo detecta que la conexión se ha cerrado y también cierra su socket.
- Puede resultar en la pérdida de datos si hay información pendiente en el proceso de envío.

3. Timeouts:
- Se establece un temporizador para cerrar automáticamente la conexión si no se recibe respuesta dentro de un cierto período de tiempo.
- Evita que la conexión permanezca abierta indefinidamente si se produce un problema de comunicación.

4. Gestión de errores:
- En caso de error, ambas partes manejan adecuadamente la situación y cierran la conexión de manera segura.
- Incluye la captura de excepciones, el registro de errores y la liberación de recursos antes de cerrar la conexión.

Métodos para evitar la pérdida de información:

1. Confirmación de recepción:
- Acuerdo entre ambas partes para confirmar la recepción de cada mensaje enviado antes de enviar el siguiente.
- Garantiza que no se pierdan mensajes y se confirme la entrega.

2. Reintento de envío:
- Si un mensaje no se confirma, el remitente puede intentar enviarlo nuevamente hasta recibir la confirmación.
- Reduce la probabilidad de pérdida de información al reenviar mensajes no confirmados.

3. Control de errores:
- Incluye mecanismos de detección y corrección de errores en los protocolos de comunicación.
- Los errores detectados pueden desencadenar el reenvío del mensaje para evitar la pérdida de información.

Al seguir un protocolo consistente y utilizar técnicas como la confirmación de recepción y el control de errores, se puede garantizar una comunicación segura y confiable, incluso al cerrar una conexión Cliente-Servidor.

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

Se realizaron los cambios necesarios para que el cliente y el servidor utilicen sockets de datagrama, es decir con implementación UDP. Los resultados fueron los siguientes:
Servidor:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h> // read(), write(), close()

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

void func(int sockfd, struct sockaddr_in cliaddr)
{
    char buff[MAX];
    int len, n;
    len = sizeof(cliaddr);
    // Loop para el chat
    for (;;) {
        bzero(buff, MAX);

        // Leo el mesaje del cliente y lo copio en un buffer
        n = recvfrom(sockfd, buff, sizeof(buff), 0, (SA*)&cliaddr, &len);
        // Muestro el buffer con los datos del cliente
        printf("Del cliente: %s\t: ", buff);
        bzero(buff, MAX);
        n = 0;
        // Copio el mensaje del servidor en el buffer
        while ((buff[n++] = getchar()) != '\n')
            ;

        // y envío el buffer al cliente
        sendto(sockfd, buff, sizeof(buff), 0, (SA*)&cliaddr, len);

        // si el mensaje dice "SALIR" salgo y cierro conexión
        if (strncmp("SALIR", buff, 4) == 0) {
            printf("Salgo del servidor...\n");
            break;
        }
    }
}

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

```
    struct sockaddr_in servaddr, cliaddr;

    // Creo el socket
    sockfd = socket(AF_INET, SOCK_DGRAM, 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);

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

    len = sizeof(cliaddr);

    // Loop infinito para atender clientes
    while (1) {
        // Recibo los datos del cliente
        func(sockfd, cliaddr);
    }

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

Cliente:

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

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

void func(int sockfd, const SA* p_servaddr, socklen_t servlen)
{
    char buff[MAX];
    int n;
    for (;;) {
        bzero(buff, sizeof(buff));
        printf("Ingrese texto : ");
        n = 0;
        while ((buff[n++] = getchar()) != '\n')
            ;
        // Envío el mensaje al servidor
        sendto(sockfd, buff, sizeof(buff), 0, p_servaddr, servlen);

        bzero(buff, sizeof(buff));
        // Recibo la respuesta del servidor
        recvfrom(sockfd, buff, sizeof(buff), 0, (struct sockaddr *) p_servaddr, &servlen);


        printf("Servidor : %s", buff);
        if ((strncmp(buff, "exit", 4)) == 0) {
            printf("Cliente cierra conexión...\n");
            break;
        }
    }
}

int main()
{
    int sockfd;
    struct sockaddr_in servaddr, cli;

    // Creo el socket y verifico
    sockfd = socket(AF_INET, SOCK_DGRAM, 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 = INADDR_ANY;
    servaddr.sin_port = htons(PORT);

    // Función para el chat
    func(sockfd, (SA*)&servaddr, sizeof(servaddr));

    // Cierro el socket
    close(sockfd);
    return 0;
}
```