Protocolos de Internet

1° CUATRIMESTRE DE 2024

Trabajo Práctico Nº 1

Grupo: 4

Profesor/a: Javier Adolfo Ouret

Integrantes:

N°	Apellido y Nombre	Legajo	Email
1	Marina Mercadal	152150976	marinamercadal@uca.edu.ar
2	Martina Naiquen Ruiz	29181	martinaruiz@uca.edu.ar
3	Carolina Suarez	152000738	suarezmacaro@gmail.com

Corrección:

Entrega 1	Devolución 1	Entrega 2	Nota

¿Qué es un Socket?

Un Socket es una forma de comunicarse con otros programas usando descriptores de archivos estándar de Unix.Un descriptor de archivos es un entero asociado a un archivo abierto.

Ese archivo puede ser una conexión de red, una cola FIFO, etc.

Para las comunicaciones en redes la llamada al sistema que se usa es Socket().

Diferentes Tipos de Sockets:

Raw sockets(sockets puros) “SOCK_RAW”

Stream sockets (sockets de flujo) “SOCK_STREAM”

Datagrama sockets (sockets de datagramas) “SPCK_DGRAM” , también llamados “sockets sin conexión”.

Los 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 en un orden dado llegaran al otro extremos en el mismo orden , sin errores.

Los Sockets de Datagrama son sockets sin conexión ya que no es necesario mantener una conexión abierta, no son confiables, si llega el datagrama los paquetes no tendrán errores, usan IP para el enrutamiento pero con UDP. Es necesario la implementación de un protocolo encima de UDP si se quiere asegurar que los paquetes lleguen.

Los Raw Sockets se utilizan cuando queremos acceder directamente a la capa de red y a la capa de enlace de datos. Permiten implementar nuevos protocolos en el espacio de usuario. Además reciben o envían el datagrama sin procesar y no incluye encabezados de nivel de enlace. SOCK_RAW incluye los encabezados IP, protocolos y datos subsecuentes.

Las estructuras necesarias para operar con sockets en el modelo C-S son:

Struct sockaddr : esta estructura mantiene información de direcciones de socket para diversos tipos de socket.

```
struct sockaddr {
unsigned short sa_family; // familia de direcciones, AF_xxx ;
char sa_data[14]; // 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.

Para manejar struct sockaddr más cómodamente, existe una estructura paralela: struct sockaddr_in (“in” por internet). Esta estructura hace más sencillo referirse a los elementos de la dirección de socket. Contiene campos para la dirección IP y el número de puerto, así como otros detalles necesarios para la comunicación a través de IPv4.

Para ingresar los datos requeridos se arman de la siguiente manera:

```
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

};
```

sin_family se corresponde con el sa_family de struct sockaddr y debe asignársele siempre el valor " AF_INET". Se utiliza para determinar que tipo de dirección contiene la estructura, asi que debe seguir orden de bytes de máquina

sin_port y sin_addr tienen que seguir el orden de bytes de la red. por lo que se necesita la siguiente estructura:

```
struct in_addr {
        unsigned long s_addr;
}
```

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().

A continuación un ejemplo de cómo se usa una estructura socket llamada my_addr con la dirección IP y el número de puerto especificados para su uso en la configuración de un socket IPv4.

```
struct sockaddr_in my_addr;
my addr.sin_family=AF_INET; //Ordenación de la red;
my addr.sin_port=htons(MYPORT);//short, para realizar la conversión a ordenación de la red desde la “ordenación de bytes de máquina”;
inet_aton (“10.12.110.57”, &(my_addr.sin_addr)); // : Convierte la dirección IP especificada ("10.12.110.57") de su formato de cadena a la representación binaria utilizada en la estructura struct sockaddr_in. La dirección IP convertida se almacena en el campo sin_addr de my_addr.
```

memset(&(my_addr.sin_zero),'\0', 8);//poner a cero el resto de la estructura para asegurar la compatibilidad con otros sistemas.

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.

En el modo bloqueante, cuando un programa hace una llamada de E/S (entrada/salida), la ejecución del programa se detiene hasta que se complete la operación de E/S. Durante este tiempo, el hilo que realizó la llamada queda inactivo. Esto significa que si hay múltiples operaciones de E/S, el programa puede quedar inactivo hasta que se completen todas ellas.

En el modo no bloqueante, cuando un programa hace una llamada de E/S, el hilo que realizó la llamada no se bloquea esperando a que se complete la operación de E/S. En cambio, el programa continúa ejecutando otras tareas o puede consultar periódicamente el estado de la operación de E/S. Esto permite que el programa realice múltiples operaciones de E/S de forma concurrente o que maneje otras tareas mientras espera que se completen las operaciones de E/S.

Los sockets calls son una llamada a una función de la API de sockets en un sistema operativo o entorno de programación que permite la comunicación de red. Estas funciones proporcionan una interfaz para crear, conectar, enviar y recibir datos a través de sockets.

Los sockets calls a los cuales se le pueden aplicar estos modos son:

En modo bloqueante:

recv(): Se bloqueará hasta que los datos estén disponibles para ser leídos del socket. Esta función recibe datos a través de un socket

send(): Se bloqueará hasta que los datos se envíen completamente a través del socket. Esta función envía datos a través de un socket.

accept(): Se bloqueará hasta que llegue una conexión entrante.

Cuando se ejecuta listen() el servidor se queda esperando a que llegue un paquete, si llamamos a recvfrom() y no hay datos el programa se bloquea en esa instrucción hasta que llegue algún dato. Las funciones como accept(), recv() y recvfrom() están configuradas por el kernel como bloqueantes al momento de crear un socket().

Para configurar en modo no bloqueante hay que llamar a fcnlt(), esto permite que un programa coloque un bloqueo de lectura o un bloqueo de escritura en un archivo.

fcntl(sockfd, F_SETFL, O_NONBLOCK);

Donde sockfd es el descriptor de archivos abierto, F_SETFL está indicando la operación a realizar, estamos configurando los flags de archivo , O_NONBLOCK es una constante que indica que el descriptor de archivos debería ser no bloqueantes.

Si se intenta leer de un socket no bloqueante y no hay datos disponibles, la función no está autorizada a bloquearse , devolverá -1 y asignará a errno el valor EWOULDBLOCK

En modo no bloqueante:

recv(): Si no hay datos disponibles para leer, esta función devolverá inmediatamente con un valor de retorno que indica que no se ha leído ningún dato.

send(): Si el socket no está listo para enviar datos (por ejemplo, si el buffer del socket está lleno), esta función devolverá inmediatamente con un valor de retorno que indica que no se han enviado datos.

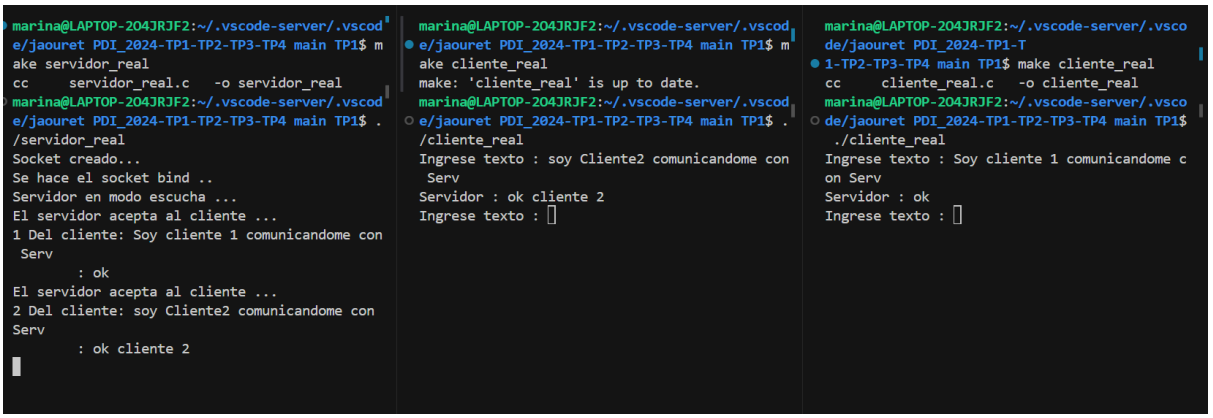
accept(): si no hay conexiones entrantes pendientes, esta función devolverá inmediatamente con un valor de retorno que indica que no se ha establecido ninguna conexión.

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

El modelo Cliente-Servidor con concurrencia Real es un modelo en donde clientes van a poder conectarse y ser atendidos por un mismo servidor al mismo tiempo.

Ejemplo con lenguaje C:

Salida en Terminal:



Archivo Cliente:

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

#define PORT 9000
#define MAX 80
#define BUFFER_SIZE 1024

void func(int sockfd) {
    char buff[MAX];
    int n;
    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;
    struct sockaddr_in servaddr;

    // Creo el socket de flujo
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Error : No se pudo crear el socket \n");
        return -1;
    }

    // Configuro la dirección del servidor
    memset(&servaddr, 0, sizeof(servaddr));
```

```
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    if (inet_pton(AF_INET, "127.0.0.1", &servaddr.sin_addr) <= 0) {
        printf("\nDirección Inválida\n");
        return -1;
    }

    // Conectarse al servidor
    if (connect(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr))
< 0) {
        printf("\nFalla en la conexión \n");
        return -1;
    }

    // Función para el chat
    func(sockfd);

    // Cierro el socket
    close(sockfd);

    return 0;
}
```

Archivo Servidor:

```
#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 <errno.h>  // errno
#define MAX 80
#define PORT 9000
#define SA struct sockaddr

void func(int connfd,int client_id) {
    char buff[MAX];
    int n;
```

```
    // Loop para el chat
    for (;;) {
        bzero(buff, MAX);

        // Leo el mensaje del cliente y lo copio en un buffer
        read(connfd, buff, sizeof(buff));
        // Muestro el buffer con los datos del cliente
        printf("%d Del cliente: %s\t: ", client_id,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
        write(connfd, buff, sizeof(buff));

        // 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, 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);
    }
    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");

    // 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);

    // Bucle principal para aceptar conexiones
    while (1) {
        connfd = accept(sockfd, (SA*)&cli, &len);
        static int client_count = 1; // Contador de clientes,
inicializado en 1
        if (connfd < 0) {
            printf("Falla al aceptar datos en el servidor...\n");
            exit(0);
        }
        else
            printf("El servidor acepta al cliente ...\n");

        // Crear un nuevo proceso hijo para manejar la conexión
        int pid = fork();
        if (pid < 0) {
            printf("Falla fork...\n");
            exit(0);
        }
        if (pid == 0) { // Proceso hijo
            close(sockfd);
            func(connfd,client_count);
            close(connfd);
            exit(0);
        } else { // Proceso padre
            close(connfd);
```

```
        }
        client_count++;
    }

    // Cierro el socket principal al salir del bucle
    close(sockfd);
    return 0;
}
```

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

La conexión Cliente- Servidor se cierra con las funciones: close() y shutdown().

Close(): cierra descriptores de archivo. Esto impedirá más lecturas y escrituras al socket. Close(sockfd)

Para más control sobre cómo se cierra el socket se puede usar la función shutdown(): int shutdown(int sockfd, int how).

donde sockfd es el descriptor de socket que se quiere desconectar y how toma el valor 0 (no se permite recibir mas datos), 1 (no se permite enviar más datos) ó 2 (no se permite enviar ni recibir más datos).

Shutdown devuelve 0 si tiene éxito y -1 en caso de error.

Cabe aclarar que shutdown() no cierra realmente el descriptor, solo cambia sus condiciones de uso. Para liberarlo hay que usar close().

Los métodos que podrían utilizarse para evitar pérdidas de información son:

Protocolos de confirmación: Implementar un protocolo de comunicación que requiera confirmaciones explícitas de la recepción de datos antes de cerrar la conexión.

Buffer de datos: para almacenar temporalmente los datos antes de ser enviados o procesados .