PROTOCOLOS DE INTERNET

1 CUATRIMESTRE DE 2024

PRÁCTICA

Trabajo Práctico N.º 1 Parte A

Comisión: FN

Profesor/a: Javier Adolfo Ouret

N°	Nombre y apellido	Carrera	DNI	Email
1	Christian Balderrama	Ing. Informática	42118900	christianbalderrama@uca.edu.ar
2	Fiorella Insfran Sanabria	Ing. Informática	96142187	fiorellainsfran@uca.edu.ar
3	Pablo Joaquin Cardozo	Ing. Informática	45178142a	cardozopabloj@uca.edu.ar
4	Pablo Joaquin Margewka	Ing. Informática	37754332	pablomargewka@uca.edu.ar
5	Fabián Leonardo De Simone	Ing. Informática	39433563	fdesimone96@uca.edu.ar

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

Un socket es una función esencial en la programación de redes que permite la comunicación entre diferentes programas mediante descriptores de archivo. En términos simples, actúa como un punto final de una conexión bidireccional entre dos programas en una red, permitiéndoles enviar y recibir datos entre sí.

- Existen varios tipos de sockets utilizados en el desarrollo de aplicaciones de red, cada uno con sus características específicas:
    - Stream Sockets (Sockets de flujo/SOCK_STREAM): Estos sockets son ideales cuando se necesita enviar datos en una secuencia específica y garantizar su llegada sin errores. Son comúnmente utilizados con el protocolo TCP (Transmission Control Protocol), proporcionando una conexión confiable y orientada a la secuencia.
    - Datagram Sockets (Sockets de datagrama/SOCK_DGRAM): Este tipo de socket ofrece un servicio de entrega de mensajes que puede no ser muy confiable, ya que se basa en datagramas individuales. Se utilizan con el protocolo UDP (User Datagram Protocol), que es útil para aplicaciones que requieren una comunicación rápida y sin conexión.
    - Raw Sockets (SOCK_RAW): Este socket proporciona acceso directo a la capa de red y transporte, permitiendo el envío y recepción de paquetes IP completos, incluidos los encabezados de los protocolos. Es útil para aplicaciones que requieren un control detallado sobre los datos de red.

Es esencial elegir el tipo de socket adecuado según los requisitos de la aplicación. Los raw sockets brindan un mayor control sobre los datos de red, mientras que los stream sockets ofrecen una comunicación confiable y ordenada, y los datagram sockets son útiles para aplicaciones que requieren una comunicación rápida y sin conexión.

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.**

Para operar con sockets en el modelo Cliente/Servidor en C, se requieren varias estructuras y funciones. A continuación se explica cómo ingresar los datos requeridos utilizando un ejemplo:

1. Creación del socket: Para crear un socket, se utiliza la función `socket()`. Esta función devuelve un descriptor de archivo que se utilizará para realizar operaciones de entrada y salida en el socket. Por ejemplo, para crear un socket TCP/IP, se utiliza la siguiente llamada:

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

Aquí, `AF_INET` especifica la familia de protocolos a utilizar (en este caso, TCP/IP) y `SOCK_STREAM` indica que se utilizará un socket de flujo.

2. Especificación de la dirección y el puerto: Antes de utilizar el socket, se debe especificar la dirección IP y el puerto de la máquina local y remota. Esto se hace utilizando una estructura llamada `sockaddr_in`. Por ejemplo, para especificar la dirección IP y el puerto de la máquina local, se puede hacer lo siguiente:

```
struct sockaddr_in local_addr;
local_addr.sin_family = AF_INET;
local_addr.sin_addr.s_addr = INADDR_ANY;
local_addr.sin_port = htons(8080);
```

Aquí, `sin_family` especifica la familia de protocolos (`AF_INET`); `sin_addr.s_addr` especifica la dirección IP de la máquina local (`INADDR_ANY` indica que se utilizará cualquier dirección IP disponible); y `sin_port` especifica el número de puerto (8080 en este caso).

3. Conexión del socket: Para establecer una conexión en un socket cliente, se utiliza la función `connect()`. Por ejemplo, para conectar el socket `sockfd` a la dirección y puerto especificados anteriormente, se puede hacer lo siguiente:

```
int status = connect(sockfd, (struct sockaddr*)&local_addr, sizeof(local_addr));
```

4. Envío y recepción de datos: Una vez establecida la conexión, se pueden enviar y recibir datos utilizando las funciones `send()` y `recv()`. Por ejemplo, para enviar datos a través del socket, se puede hacer lo siguiente:

```
int bytes_sent = send(sockfd, buffer, sizeof(buffer), 0);
```

Aquí, `buffer` es un arreglo de bytes que contiene los datos a enviar.

5. Cierre del socket: Después de finalizar la comunicación, es importante cerrar el socket utilizando la función `close()`. Por ejemplo, para cerrar el socket `sockfd`, se puede hacer lo siguiente:

```
close(sockfd);
```

Al seguir estos pasos y corregir los detalles mencionados, se podrá operar con sockets en el modelo Cliente/Servidor en C de manera efectiva.

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: Sockets en modo bloqueante, las llamadas de sistema asociadas pueden dejar el proceso en espera hasta que la operación solicitada se complete. Por ejemplo, si realizas una llamada a recv(), accept(), o recvfrom() y no hay datos disponibles para recibir, tu programa quedará en espera hasta que lleguen los datos o se complete la operación.

- Funciones que admiten bloqueo:
    - accept(): Espera y acepta una conexión entrante.
    - recv(): Recibe datos de un socket.
    - recvfrom(): Recibe datos de un socket específico, incluyendo la dirección del remitente.

Modo No Bloqueante: En cambio, en el modo no bloqueante, las llamadas de sistema retornan inmediatamente, incluso si la operación solicitada no puede completarse de inmediato. Por ejemplo, si no hay datos disponibles para recibir, recv() puede retornar inmediatamente con un valor de error o un indicador especial, en lugar de bloquear el proceso.

- Funciones que admiten no bloqueo:
    - Todas las funciones de socket mencionadas anteriormente (accept(), recv(), recvfrom()) pueden ser configuradas para operar en modo no bloqueante. Para esto, debes configurar opciones especiales en el socket usando la función fcntl().

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

El modelo Cliente - Servidor, establece que en todo par de aplicaciones que se comunican, una de ellas (el servidor) debe iniciar su ejecución y permanecer en modo de espera, hasta que llegue una solicitud por parte de la otra aplicación (el cliente).

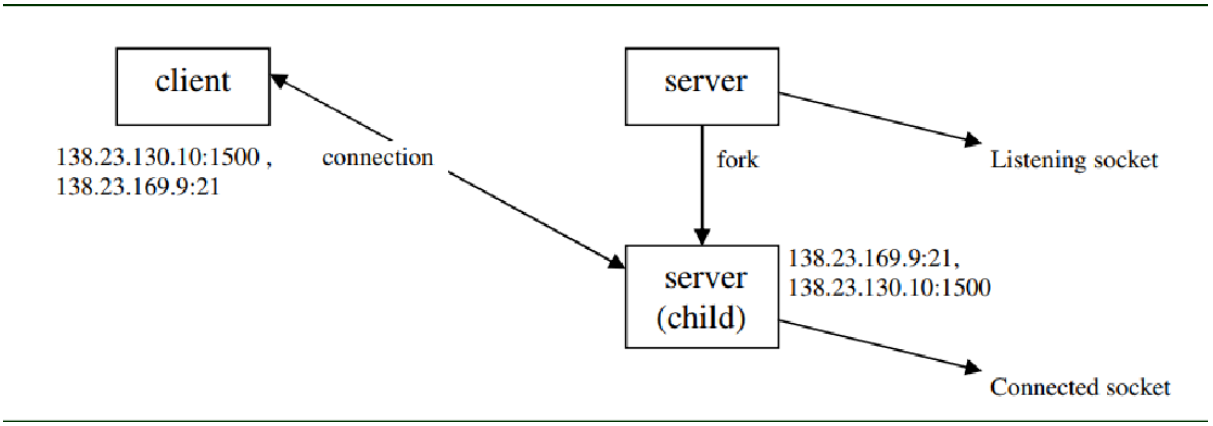
Este modelo permite establecer la comunicación entre ambas aplicaciones y que puedan pasar datos entre ellos.

En el caso de un servidor con concurrencia real, cuando el servidor recibe y acepta la conexión con el cliente, crea una copia de sí mismo llamado proceso hijo y delega en el hijo la comunicación con el cliente.

Esta bifurcación se realiza a través de una función llamada fork().

La ventaja de esto, es que ahora el servidor puede aceptar múltiples conexiones. En caso de que se conecte un nuevo cliente, nuestro servidor hará un nuevo fork, creando un nuevo proceso hijo que se hará cargo de la comunicación con el nuevo cliente. Tanto el padre como los procesos hijos pueden diferenciarse a través del número de PID.

Al poder aceptar múltiples conexiones al mismo tiempo, el servidor puede responder a las solicitudes de una manera más rápida, mejorando el rendimiento, la escalabilidad y tener un tiempo de respuesta más reducido.



En la siguiente imagen podemos observar como el servidor creó un proceso hijo que se conectó con el cliente, y por otro lado mantiene en el proceso padre el listening para poder aceptar más conexiones.

Como ejemplo de servidor concurrente tenemos un servidor que recibe números enteros y devuelve al cliente el factorial de ese número.
Primero corrimos el servidor desde la terminal y luego el cliente

A continuación mostramos la salida del servidor:

```
pjcdz@laptop:~/Documents/GitHub/ProtocolosDeInternet-UCA-2024/c041
5/TP1-Clase1504$ sudo ./EJ01_01_Socket_Servidor_Concurrente
Socket creado...
Se hace el socket bind ..
Servidor en modo escucha ...
El servidor acepta al cliente ...
Número recibido: 1
Número recibido: 2
Número recibido: 3
Número recibido: 4
Número recibido: 5
El cliente ha cerrado la conexión.
```

Luego, una parte de salida del cliente:

```
pjcdz@laptop:~/Documents/GitHub/ProtocolosDeInternet-UCA-2024/c041
5/TP1-Clase1504$ sudo ./EJ01_01_Socket_Cliente_B
Socket creado ..
Conectado al servidor..
PID del proceso padre: 15467

Enviado: 1 - Factorial de 1: 1
Tiempo de ejecución: 178 nanosegundos
PID del proceso hijo: 15471

Enviado: 2 - Factorial de 2: 2
Tiempo de ejecución: 138 nanosegundos
PID del proceso hijo: 15471

Enviado: 3 - Factorial de 3: 6
Tiempo de ejecución: 166 nanosegundos
PID del proceso hijo: 15471

Enviado: 4 - Factorial de 4: 24
Tiempo de ejecución: 146 nanosegundos
PID del proceso hijo: 15471

Enviado: 5 - Factorial de 5: 120
Tiempo de ejecución: 124 nanosegundos
PID del proceso hijo: 15471
```

El cliente envía números al servidor y recibe la respuesta del servidor, que la imprime en pantalla. Lo más importante es que se puede ver que el servidor creó un fork() para atender esta solicitud del cliente. Al crear un fork() el servidor crea una

copia de sí mismo que se va a encargar de realizar la solicitud del cliente. Este nuevo proceso hijo tiene un número de PID que es diferente al del padre.

El codigo usado fue el siguiente:

Servidor:

```
#include <arpa/inet.h> // inet_addr()
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // bzero()
#include <sys/socket.h>
#include <unistd.h> // read(), write(), close()
#include <time.h> // clock_gettime()
#include <sys/types.h> // pid_t

// sudo ./EJ01_01_Socket_Servidor_Concurrente

#define MAX 1024
#define PORT 6667
#define SA struct sockaddr

double tiempo_transcurrido(struct timespec *inicio, struct timespec
*fin) {
    return (fin->tv_sec - inicio->tv_sec) * 1e9 + (fin->tv_nsec -
inicio->tv_nsec);
}

void func(int* sockfd) {
   struct timespec start, end;
   int connfd = *sockfd;
   int buff;
   for (;;) {
       int numBytes = read(connfd, &buff, sizeof(int));
       if (numBytes <= 0) {
           printf("El cliente ha cerrado la conexión.\n");
           break;
       }
       printf("Número recibido: %d\n", buff);

       // Calcular el factorial
       clock_gettime(CLOCK_MONOTONIC, &start);
```

```
        int factorial = 1;
        for(int i = 1; i <= buff; ++i) {
            factorial *= i;
        }
        clock_gettime(CLOCK_MONOTONIC, &end);

        // Obtener el tiempo de ejecución
        double execution_time = tiempo_transcurrido(&start, &end);

        // Enviar la respuesta al cliente
        char response[MAX];
          sprintf(response, "Factorial de %d: %d\nTiempo de ejecución:
%.0f  nanosegundos\nPID  del  proceso  hijo:  %d\n",  buff,  factorial,
execution_time, getpid());
        write(connfd, response, sizeof(response));

        if (strncmp("SALIR", response, 4) == 0) {
            printf("Salgo del servidor...\n");
            break;
        }
    }
    close(connfd);
    exit(0);
}

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

    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));

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORT);

    if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
```

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

    if ((listen(sockfd, 5)) != 0) {
        printf("Falla el listen ...\n");
        exit(0);
    }
    else
        printf("Servidor en modo escucha ...\n");

    while (1) {
        unsigned int len = sizeof(cli);
        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 ...\n");

        int pid = fork();
        if (pid < 0) {
            printf("Falla al crear el proceso...\n");
        }
        else if (pid == 0) { // Proceso hijo
                close(sockfd); // Cerrar el descriptor de archivo del
servidor en el proceso hijo
            func(&connfd);
        }
        else { // Proceso padre
            char response[MAX];
            sprintf(response, "PID del proceso padre: %d\n", getpid());
            write(connfd, response, sizeof(response));
            close(connfd); // Cerrar el socket en el proceso padre
        }
    }

    close(sockfd);
}
```

Cliente:

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

// sudo ./EJ01_01_Socket_Cliente_B

#define MAX 1024
#define PORT 6667
#define SA struct sockaddr

void func(int sockfd) {
   char buffer[MAX] = {0};
   int leovalor;

   for (int i = 1; i <= 5; i++) {  // Cambiado de 10 a 5
       // Enviar número al servidor
       write(sockfd, &i, sizeof(int));

       // Leer respuesta del servidor
       leovalor = read(sockfd, buffer, MAX);
       printf("%s\n", buffer);

       printf("Enviado: %d - ", i);
   }

   // Leer respuesta final del servidor
   leovalor = read(sockfd, buffer, MAX);
   printf("%s\n", buffer);
}

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

   // 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");

    // Función para el chat
    func(sockfd);

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

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

La conexión cliente-servidor se puede cerrar utilizando los métodos `close()` y `shutdown()`.

La función `close()` se utiliza para cerrar el socket que se está utilizando, lo que impide enviar o recibir más datos a través de ese socket. Su prototipo es `int close(int fd)`, donde `fd` es el descriptor de archivo.

Por otro lado, la función `shutdown()` se emplea para cambiar las condiciones de uso del socket, no necesariamente para cerrar la conexión. Al especificar el parámetro `how`, se puede controlar si se permite recibir más datos, enviar más datos o ambas acciones. Los valores posibles para `how` son:
- 0: No se permite recibir más datos.
- 1: No se permite enviar más datos.
- 2: No se permite enviar ni recibir más datos.

Es importante destacar que `shutdown()` no cierra el socket, solo modifica sus condiciones de uso. Para cerrar definitivamente un socket y liberar recursos, se debe utilizar la función `close()` [T4].

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

Al modificar el tipo de socket que estaba en SOCK_STREAM a SOCK_DGRAM vamos a tener varias diferencias. El SOCK_DGRAM no está orientado a la conexión y los paquetes se envían sin recibir una confirmación por parte del cliente. Además los paquetes pueden llegar desordenados o perderse en el camino. Como estamos probando el servidor y el cliente desde la misma máquina local, todos los paquetes van a llegar ordenados y no va a haber pérdida de paquetes. Por lo tanto, desde el punto de vista de la ejecución el cambio de socket va a resultar imperceptible.
Ahora, desde el punto de vista del código, al cambiar a SOCK_DGRAM se eliminan algunas funciones innecesarias como listen(), accept() y connext().