UCA
Pontificia Universidad Católica Argentina

TP 1 – Parte A

Integrantes: Lucas Debarbieri, Gonzalo Crucitta, Andres Luza, Sebastian Lernoud

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

Un socket es un puntero a una estructura de datos compuesta por un file descriptor, una IP y puerto de origen y destino, y una familia o dominio de conexión junto con los protocolos que soporta. Es un conector entre proceso que se utiliza para que los programas se comuniquen mediante el uso de archivos descriptores Unix. Los diferentes tipos de sockets son:

- Raw sockets (sockets puros) "SOCK_RAW": el concepto básico de sockets a bajo nivel es enviar un solo paquete por vez con todos los encabezados de los protocolos completados dentro del programa en lugar de usar el kernel. Este tipo de socket incluye los encabezados IP, protocolos y datos subsecuentes.
- Stream sockets (sockets de flujo) "SOCK_STREAM": definen flujos de comunicación en dos direcciones, confiables y con conexión TCP, e IP para el enrutamiento.
- Datagram sockets (sockets de datagramas) "SOCK_DGRAM": son sockets sin conexión porque no es necesario mantener una conexión abierta como con los sockets de flujo. Usan IP para el enrutamiento, pero con UDP. Simplemente se arma un paquete, se le coloca una cabecera IP con la información de destino y se envía. Es necesario implementar un protocolo encima de UDP si se quiere asegurar que los paquetes lleguen.

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

- struct sockaddr: mantiene información de direcciones de socket para diversos tipos de socket.
- struct sockaddr_in: hace más sencillo referirse a los elementos de la dirección de socket.
- Struct in_addr: dirección de internet.

Para ingresar los datos requeridos, primero tenemos que crear una variable del tipo struct sockaddr_in y luego cargar los datos de las variables dentro de la estructura individualmente.

```
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET; // Ordenación de máquina
my_addr.sin_port = htons(MYPORT); // short, Ordenación de la 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.**

En el modo bloqueante 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. Si fuese una función no bloqueante se puede preguntar al socket por su estado. 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).

Los “sockets calls” a los que se puede aplicar estos modos son:
• accept()
• recv()
• recvfrom()

La función necesaria es select() que da la posibilidad de comprobar varios sockets al mismo tiempo, indica cuáles están listos para leer, cuáles para escribir y cuáles han generado excepciones. Es una herramienta para monitorear multiples sockets y evitar que el programa se bloquee al esperar a que se completen las operaciones de entrada/salida.

En el modo no bloqueante, la función select() se utiliza para evitar que las funciones recv() o send() bloqueen la ejecución del programa si no hay datos disponibles en el socket. Después de que se crea el socket y se establece su configuración como no eloquent, se pueden usar las funciones connect(), recv y send() de forma similar al modo bloqueante, pero las funciones de entrada/salida pueden devolver un valor error.
La función select() se encarga de indicar al kernel cuanto tiempo debe esperar para ejecutar un evento y antes de pasar al siguiente, como por ejemplo si hay descriptores listos para lectura y si pasa un determinado tiempo.
Los parámetros de dicha función son:
• Estructura timeval
• Readfds / Writefds / Exceptfds
• Conjuntos (set del fds)

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

El modelo C-S aplicado a un servidor con concurrencia real se puede describer de la siguiente manera. El servidor tiene la capacidad de manejar o proveer servicios a varios clientes usando forking o threads en sistemas de multiples procesadores. Los procesadores se asignan a distintos procesos. En el forking, los procesos y el codigo se separan por el proceso padre y varios hijos, cada uno con su propio PID (process ID). Cada cliente esta conectado al servidor con un proceso hijo, dejando que deja que multiples clientes esten conectados a un mismo servidor, sin interrumpir el servicio a cada uno y a su vez dejando que el servidor pueda manejar otros requests. Con threading, dentro de un mismo proceso se usa la misma memoria pero con diferentes estados.

Ejemplo de codigo:

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

// adaptado para funcionamiento con mac, no estoy seguro si corre igual en Linux/Windows!

#define MYPORT 3490
#define BACKLOG 10

int main()
{
    int sockfd, new_fd;
    struct sockaddr_in my_addr;
    struct sockaddr_in their_addr;
    socklen_t sin_size;
    printf("Esta en el socket: %d\n", socket(AF_INET, SOCK_STREAM, 0));

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == 1)
    {
        perror("socket");
        exit(1);
    }

    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(MYPORT);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 8);

    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1)
    {
        perror("bind");
        exit(1);
    }

    if (listen(sockfd, BACKLOG) == -1)
    {
        perror("listen");
        exit(1);
    }
```

```
    while (1)
    {
        sin_size = sizeof(struct sockaddr_in);
        if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1)
        {
            perror("accept");
            continue;
        }

        printf("server: got connection from %s\n", inet_ntoa(their_addr.sin_addr));

        if (!fork())
        { // proceso hijo
            if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }

        close(new_fd); // proceso padre

        while (waitpid(-1, NULL, WNOHANG) > 0)
            ;
    }

    return 0;
}
```

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

La conexión C-S se puede cerrar mediante close() y shutdown().

• close(): Cierra de forma abrupta los descriptores de archivos, es decir, no permite más lecturas ni escrituras al socket.

• shutdown(): no cierra el descriptor, sino que cambia las condiciones para dejar de recibir o para dejar de enviar. Con este método se evita la pérdida de información.

Para evitar a perdida de información se puede utilizar diferentes técnicas o métodos:

- Enviar un mensaje de cierre: Este se da desde el lado que está cerrando la conexión de que se cierre. De esta manera, el otro lado puede recibir el mensaje y cerrar la conexión de manera adecuada.
- Usar timeouts: Otra técnica es usar timeouts en la comunicación para asegurarse de que si no hay actividad en un período determinado de tiempo, se cierre la conexión. Esto permite que cualquier información pendiente se transmita antes de que se cierre la conexión.

- Usar técnicas de confirmación de recepción: Esto implica que el receptor de la información envía un mensaje de confirmación después de recibir cada paquete de datos, lo que garantiza que se transmita correctamente antes de cerrar la conexión.
- Usar técnicas de retransmisión de datos: Esto implica reenviar los datos que no se recibieron correctamente o que se perdieron debido a la desconexión.

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

En este punto usamos y modificamos un poco el código que se encontraba en el Github para que se comuniquen dos sockets, uno siendo el cliente y otro el servidor. La comunicacion se da a traves de sockets TCP donde previamente se conectan entre ellos.

```
Socket Creado
socket bind Completado
socket en modo escucha - pasivo
conexion con ('127.0.0.1', 15080).
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b'1234567890'"
enviando mensaje de vuelta al cliente
recibido "b''"
no hay mas datos ('127.0.0.1', 15080)
```

.

```
conectando a %s puerto %s ('localhost', 6667)
Conectando a localhost puerto 6667
Paso: 0
enviando "b'123456789012345678901234567890'"
recibiendo "b'1234567890'"
recibiendo "b'1234567890'"
recibiendo "b'1234567890'"
Paso: 1
enviando "b'123456789012345678901234567890'"
recibiendo "b'1234567890'"
recibiendo "b'1234567890'"
recibiendo "b'1234567890'"
Paso: 2
enviando "b'123456789012345678901234567890'"
recibiendo "b'1234567890'"
recibiendo "b'1234567890'"
recibiendo "b'1234567890'"
Paso: 3
enviando "b'123456789012345678901234567890'"
recibiendo "b'1234567890'"
recibiendo "b'1234567890'"
recibiendo "b'1234567890'"
Paso: 4
enviando "b'123456789012345678901234567890'"
recibiendo "b'1234567890'"
recibiendo "b'1234567890'"
recibiendo "b'1234567890'"
Paso: 5
enviando "b'123456789012345678901234567890'"
recibiendo "b'1234567890'"
recibiendo "b'1234567890'"
recibiendo "b'1234567890'"
cerrando socket
```

Podemos ver entonces del lado izquierdo el Socket Servidor que espera la conexion del cliente y una vez que recibe los datos, los devuelve. Cada paso enunciado en la pantalla del cliente es cada envío del string que enuncia del 1 al 0 tres veces.
Para continuar con el ejercicio cambiamos ambos sockets a sockets de datagramas, lo que genero que cambiemos la lógica de ambos programas.

```
Socket Creado
socket bind Completado
Mensaje recibido de ('127.0.0.1', 55597): 1234567890123456789012345678 90
Mensaje recibido de ('127.0.0.1', 55597): 1234567890123456789012345678 90
Mensaje recibido de ('127.0.0.1', 55597): 1234567890123456789012345678 90
Mensaje recibido de ('127.0.0.1', 55597): 1234567890123456789012345678 90
Mensaje recibido de ('127.0.0.1', 55597): 1234567890123456789012345678 90
Mensaje recibido de ('127.0.0.1', 55597): 1234567890123456789012345678 90
Ingrese 0 si quiere finalizar: 0
cerrando socket
```

```
conectando a 127.0.0.1 puerto 6667
Paso: 1
enviando "b'1234567890123456789012345678 90'"
Respuesta del servidor UDP: 1234567890123456789012345678 90
Paso: 2
enviando "b'1234567890123456789012345678 90'"
Respuesta del servidor UDP: 1234567890123456789012345678 90
Paso: 3
enviando "b'1234567890123456789012345678 90'"
Respuesta del servidor UDP: 1234567890123456789012345678 90
Paso: 4
enviando "b'1234567890123456789012345678 90'"
Respuesta del servidor UDP: 1234567890123456789012345678 90
Paso: 5
enviando "b'1234567890123456789012345678 90'"
Respuesta del servidor UDP: 1234567890123456789012345678 90
Paso: 6
enviando "b'1234567890123456789012345678 90'"
Respuesta del servidor UDP: 1234567890123456789012345678 90
cerrando socket
```

Para que ambos se comuniquen fue necesario hacer unos cambios, de todas formas, se puede lograr que tengan un mismo intercambio de mensajes a pesar de la modificación en el tipo de sockets.

```python
import socket
import threading

direccion_servidor = ("localhost",6667)
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
print ("Socket Creado")
sock.bind(direccion_servidor)
print ("socket bind Completado")
#sock.listen(1)


continuar = 1
contador = 1
while continuar:
    # Recibir datos y dirección del cliente
    data, client_address = sock.recvfrom(30)
    print(f"Mensaje recibido de {client_address}: {data.decode()}")

    # Enviar una respuesta al cliente
    response = "Hola desde el servidor UDP"
    sock.sendto(data, client_address)
    if contador==6:
        continuar = int(input("Ingrese 0 si quiere finalizar: "))
    contador+=1

print ('cerrando socket')
sock.close()
```