PDI - TP 1 - A (Entrega 22/04/2024)
Profesor: Ing. Javier A. Ouret.
Alumnos: Ivan Bachmann, Ignacio Di Bartolo, Ian Capon, Franco Reno.

Explicar qué es un socket y los diferentes tipos de sockets.
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.
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.
Describir el modelo C-S aplicado a un servidor con Concurrencia Real. Escribir un ejemplo en lenguaje C.
Cómo se cierra una conexión C-S ?. Métodos que eviten la pérdida de informacion.
Probar el código del chat entre cliente y servidor. Cambiar tipo de socket y volver a probar.
Presentar las respuestas en archivos ,pdf identificando alumno y grupo.
C-S = Cliente – Servidor
RESPUESTAS
Un socket determina los puntos de conexión origen y destino entre un cliente y un servidor que se comunican a traves de la red utilizando TCP/IP. Esta compuesto por IP (del host) + Puerto(Identifican aplicaciones). El socket funciona como E/S de un archivo; es una llamada al sistema que devuelve un descriptor de comunicaciones.
Tipos de sockets;
- RAW SOCKETS(sockets puros);
- STREAM SOCKETS(sockets de flujo); Son sockets de comunicación en dos direcciones. Los paquetes llegan en orden y utilizan conexión TCP, por lo que llegan sin errores y por un medio confiable.
- DATAGRAM SOCKETS(sockets de datagrama); Son sockets sin conexión, como utilizan UDP, no es fiable que llegue a destino aunque si es que llegan, no tendran errores. No es necesario tener una conexión abierta para la comunicación.
Para crear sockets se necesita primero construir un “descriptor de socket” la cual es una estructura que mantiene informacion de direcciones, estas estructuras son las siguientes:

1) Estructura para familia de direcciones y numero de puerto:
struct sockaddr {

```
        unsigned short sa_family;   //familia de direcciones, AF xxx
        char sa_data[14]                    //14 bytes de direccion de protocolo
        };
```

2) Estructura para referirse a los elementos de la direccion de socket:

```
struct sockaddr_in{
        short int sin_family;
        unsigned short int sin_port;
        struct in_addr sin_addr;
        unsigned char sin_zero[8];                  //reserva size del struct sockaddr
}

//direccion de internet
struct in_addr{
        unsigned long s_addr;       //long de 64 bits, 32,...
}
```

+ MODO BLOQUEANTE:

Si no hay datos que lleguen al servidor luego de llamar a “recvfrom”, las demas funciones se bloquean hasta que se reanude el flujo de datos. Las funciones que permiten el bloqueo desde el Kernel son: “recv()”, “recvfrom()”, “accept()”.

Para evitar que una funcion sea bloqueante se debe llamar a la funcion “fcntl()”; este es el punto de acceso para varias operaciones de descriptores de archivos. La llamada a este funcion permite que un programa coloque un bloqueo de escritura o de lectura en un archivo.

+ MODO NO BLOQUEANTE:

Para hacer que un socket no sea bloqueante, se llama a “fcntl()”; Al establecer un socket no bloqueante se puede preguntar por su estado y si se intenta acceder a un no bloqueante sin datos, este no puede quedar como bloqueante, devuelve un -1. Para resolver esto, y el socket no este en espera de datos en un bucle, se coloca la funcion “select()”.

Cliente – servidor con Concurrencia Real

Un servidor con concurrencia real es capaz de atender multiples solicitudes de clientes en simultaneo: para esto, es necesario replicar multiples procesos para responder

a esas conexiones con los clientes, por lo que el servidor creara varios procesos hijos a traves de la funcion “fork()”. Esta funcion, replica al servidor padre para atender una solicitud de conexión. Después de crear un nuevo proceso hijo, ambos procesos ejecutarán la siguiente instrucción después de la llamada al sistema fork ().

Prueba en C:

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

#define PORT 8080
#define MAX_CLIENTS 5

void *call_cliente(void *arg) {
    int client_socket = *((int *)arg);
    char buffer[1024] = {0};
    char *respuesta = "Mensaje recibido.\n";

    // Recibo el mensaje del cliente
    recv(client_socket, buffer, 1024, 0);
    printf("Mensaje del cliente: %s\n", buffer);

    // Envio de respuesta al cliente
    send(client_socket, response, strlen(response), 0);

    close(client_socket);
}

int main() {
    int server_fd, new_socket, client_sockets[MAX_CLIENTS], client_count = 0;
    struct sockaddr_in address;
```

```
    int opt = 1;
    int addrlen = sizeof(address);
    pthread_t tid[MAX_CLIENTS];

    // Creo el socket del servidor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }
    // Configuracion socket address:
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    // Bind
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    // Listening
    if (listen(server_fd, MAX_CLIENTS) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    while (1) {
        // Acepto conexiones hasta error
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen)) < 0) {
            perror("accept");
            exit(EXIT_FAILURE);
        }
```

```
        // Creo un thread para atender al cliente
        pthread_create(&tid[client_count], NULL, call_cliente, &new_socket);

        // Creo una copia donde almaceno a los clientes
        client_sockets[client_count] = new_socket;
        client_count++;
    }

    // cuento todos los threads
    for (int i = 0; i < client_count; i++) {
        pthread_join(tid[i], NULL);
    }

    return 0;
}
```

Bibliografía a consultar:

Clases Teóricas.

Brian "Beej Jorgensen" Hall. Beej's Guide to Network Programming: Using Internet Sockets. 2019.
Douglas E. Comer, David L. Stevens. Internetworking With Tcp/Ip: Client-Server Programming and Applications. Prentice Hall. 2011.

PDI - TP 1 - B - Raw Sockets

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.

Unix provee dos tipo de sockets para el acceso directo a la red:

SOCK_PACKET, recibe y envía datos al dispositivo ubicado en la capa de enlace. Esto significa que el "header" de la placa de red incluido en los datos puede ser escrito o leído. En general es el header de Ethernet. Todos los encabezados de los protocolos subsecuentes también deben ser incluidos en los datos.

SOCK_RAW, es el que usaremos por ahora, que incluye los encabezados IP, protocolos y datos subsecuentes.

Desde el momento que lo creamos se puede enviar cualquier tipo de paquetes IP por este socket. También se pueden recibir cualquier tipo de paquetes que lleguen al host, después que el socket fue creado, si se hace un "read()" desde él. Se puede observar que aunque el socket es una interfaz al header IP, es tambiés específico para una capa de transporte. Esto significa que para escuchar tráfico TCP, UDP, ICMP hay que crear 3 raw sockets por separado, usando IPPROTO_TCP, IPPROTO_UDP y IPPROTO_ICMP (números de protocolo son 0 ó 6 para tcp, 17 para udp y 1 para icmp).

Con esta información es posible crear un simple "sniffer", que muestre todo el contenido de los paquetes TCP que se reciban. (En este ejemplo se evitan los headers IP y TCP, y se imprime solamente el "payload" con encabezados IP y TCP contenidos en el paquete).

- Utilizando el código raw.c como base escribir un "sniffer" que es un programa que muestra el contenido del tráfico que llega.
- Enviar tráfico al "sniffer" desde el cliente escrito en la parte A del TP1
- Enviar tráfico ICMP al "sniffer" y mostrar los resultados del LOG con comentarios.
- Mostrar resultados.

PDI - TP 1 - C - Cliente Servidor utilizando Sockets en Python.

Para realizar programas Cliente Servidor con Python utlizamos la librería o paquete socket.py (https://github.com/python/cpython/blob/3.10/Lib/socket.py)). Esta librería es una transcripción sencilla de la llamada al sistema sockets de BSD Unixal estilo orientado

a objetos de Python:. La función socket() devuelve a socket object métodos que implementan las diversas llamadas al sistema de socket. Los tipos de parámetros tienen un nivel algo más alto que en la interfaz C, como con read() y write() en el uso de los archivos Python, la asignación del buffer es automática y la longitud del buffer está implícita en las operaciones de envío.
Para detalles de la implementación de socket.py ver;
Para la parte C del trabajo práctico usar como ejemplos de base los siguientes códigos (la explicación de la implementación está detallada dentro del mismo código)
https://trello.com/1/cards/64184f125aa3494c371fb5f6/attachments/6445b7a56208701889f75156/download/image.png
y escribir una aplicación cliente servidor que muestre las direcciones y puertos de todos los clientes conectados del lado del servidor y devuelva a cada cliente el día y hora de conexión, y el tiempo que estuvo (o está conectado).
Realizar la misma aplicación tanto para C-S con Concurrencia Aparente (Select) como C-S Concurrente.
De ser necesario agregar tiempo de espera, loop, sleep, con contadores para demorar los procesos.
Implementar una limpieza de recuersos al salir del los programas (agregar opción de pregunta al usuario para cerrar los clientes).
Para crear un socket (stream) en Python: socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)
Los parámetros son los mismos que se usan en C

```
import socket
Creo socket IPv4
sock_fd = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
if sock_fd == -1:

Administro el error

Ejemplo de cliente sencillo

from socket import socket as Socket
from socket import AF_INET, SOCK_STREAM
SERVIDOR = 'a.b.c.d'  # IP
```

```
NROPUERTO = 41267          # puerto
BUFFER = 80                # tamaño del buffer

DIRECCION_SERVIDOR = (SERVIDOR, NROPUERTO)
CLIENTE = Socket(AF_INET, SOCK_STREAM)
try:
    CLIENTE.connect(SERVER_ADDRESS)
    print('cliente conectado')
    DATOS = input('Mensaje : ')
    CLIENTE.send(DATOS.encode())
except OSError:
    print('connection failed')
CLIENT.close()
```