Ejercicios del 15/4

1. **Modificar el código del programa Cliente Servidor Concurrente para que el cliente genere bucles de mensajes con el siguiente proceso y que el servidor devuelva los tiempos de ejecución y PID de hijos y padres asociados al clente. El proceso que debe realizar el servidor es el cálculo del factorial del nro que le enviamos desde el cliente.**

cliente_base.c:

```c
#include <arpa/inet.h> // inet_addr()
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h> // bzero()
#include <sys/socket.h>
#include <unistd.h> // read(), write(), close()
#define MAX 80
#define PORT 8080
#define SA struct sockaddr
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, connfd;
    struct sockaddr_in servaddr, cli;

    // 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);
}
```

Servidor_Concurrente_Mod_3:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```c
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/time.h>

#define PORT 8080
#define FALLA 10
#define EPROCESO_EXITOSO 0

unsigned long long factorial(int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}

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

    printf("Proceso hijo (PID: %d) establecido para el cliente\n",
pid);

    // Enviamos un mensaje al cliente
    sprintf(buffer, "Servidor: Conectado con cliente, PID Padre: %d,
PID Hijo: %d\n", getppid(), getpid());
    send(conn_sock, buffer, strlen(buffer), 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;
        }

        int numero = atoi(buffer); // Convertir el número recibido a
entero
        printf("Recibido: %d\n", numero);

        // Calcular el factorial del número recibido
```

```
        struct timeval start, end;
        gettimeofday(&start, NULL);
        unsigned long long result = factorial(numero);
        gettimeofday(&end, NULL);

        long microseconds = ((end.tv_sec * 1000000 + end.tv_usec) -
(start.tv_sec * 1000000 + start.tv_usec));
        printf("Tiempo de cálculo del factorial: %ld microsegundos\n",
microseconds);

        // Convertir el resultado y el tiempo a una cadena y enviar al
cliente
        sprintf(buffer, "Factorial: %llu, Tiempo de cálculo: %ld
microsegundos\n", result, microseconds);
        send(conn_sock, buffer, strlen(buffer), 0);
    }

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

int main()
{
    int sockfd, nuevo_socket;
    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)
    {
        // Aceptar una nueva conexión de la cola del listen
        if ((nuevo_socket = accept(sockfd, (struct sockaddr
*)&cli_addr, &cli_len)) < 0)
        {
            perror("Falla accept");
            exit(FALLA);
        }

        pid_t pid = fork();
        if (pid < 0)
        {
            perror("Falla fork");
            exit(FALLA);
        }
        if (pid == 0)
        {                                          // Proceso hijo
            close(sockfd);                         // Cerrar el
descriptor de archivo del servidor en el proceso hijo
            proceso_hijo(nuevo_socket, getpid()); // Manejar la
conexión con el cliente
            exit(EPROCESO_EXITOSO);
        }
        else
        { // Proceso padre
            printf("Proceso padre (PID: %d) establecido para el
cliente\n", getpid());
```

```
            close(nuevo_socket); // Cerrar el socket de conexión en el
proceso padre
        }
    }

    // Cerramos el socket del servidor (no se alcanza aquí, pero por
completitud)
    close(sockfd);

    return 0;
}
```

2. **Al código anterior experimentar con fork() del lado cliente para que genere varios clientes simultáneos en el servidor. La IP origen será la misma pero cada proceso debe tener números de puerto diferentes.**

Cliente_fork_3.c:

```
#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>
#include <sys/wait.h>

#define MAX 80
#define PORT 8080
#define NUM_CONNECTIONS 5
#define SA struct sockaddr

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;

    // 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");

    // Crear múltiples conexiones utilizando fork()
    for (int i = 0; i < NUM_CONNECTIONS; ++i)
```

```
    {
        pid_t pid = fork();
        if (pid == -1)
        {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        else if (pid == 0)
        { // Proceso hijo
            printf("Proceso hijo %d creando conexión...\n", getpid());

            // Función para el chat
            func(sockfd);

            // Cerrar el socket y terminar el proceso hijo
            close(sockfd);
            exit(EXIT_SUCCESS);
        }
    }

    // Esperar a que todos los procesos hijos terminen
    for (int i = 0; i < NUM_CONNECTIONS; ++i)
    {
        wait(NULL);
    }

    // Cerrar el socket del padre
    close(sockfd);
    return 0;
}
```

3. **Compilar y ejecutar los ejemplos con SOCKRAW. Modificar los ejemplos para hacer loopback como se muestra en el código con loopback.**

raw_socket_loopback.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netinet/tcp.h>
```

```c
#include <arpa/inet.h>
#include <net/ethernet.h>
#include <netpacket/packet.h> // Agregar esta línea
#include <net/if.h>

#define BUFFER_SIZE 65536

void procesar_paquete(unsigned char *buffer, int size) {
    printf("Paquete recibido - Longitud: %d bytes\n", size);

    // Encabezado Ethernet
    struct ethhdr *encabezado_eth = (struct ethhdr *)buffer;
    printf("Encabezado Ethernet\n");
    printf("  Dirección MAC de origen:
%.2X:%.2X:%.2X:%.2X:%.2X:%.2X\n",
           encabezado_eth->h_source[0], encabezado_eth->h_source[1],
encabezado_eth->h_source[2],
           encabezado_eth->h_source[3], encabezado_eth->h_source[4],
encabezado_eth->h_source[5]);
    printf("  Dirección MAC de destino:
%.2X:%.2X:%.2X:%.2X:%.2X:%.2X\n",
           encabezado_eth->h_dest[0], encabezado_eth->h_dest[1],
encabezado_eth->h_dest[2],
           encabezado_eth->h_dest[3], encabezado_eth->h_dest[4],
encabezado_eth->h_dest[5]);
    printf("  Tipo de protocolo: 0x%.4X\n",
ntohs(encabezado_eth->h_proto));

    // Encabezado IP
    struct iphdr *encabezado_ip = (struct iphdr *)(buffer +
sizeof(struct ethhdr));
    printf("Encabezado IP\n");
    printf("  Versión IP: %d\n", encabezado_ip->version);
    printf("  Longitud del encabezado IP: %d bytes\n",
encabezado_ip->ihl * 4);
    printf("  Tipo de servicio: %d\n", encabezado_ip->tos);
    printf("  Longitud total: %d bytes\n",
ntohs(encabezado_ip->tot_len));
    printf("  Identificación: %d\n", ntohs(encabezado_ip->id));
    printf("  Fragmentación: Flags: %d, Offset: %d\n",
(encabezado_ip->frag_off & 0x1FFF), (encabezado_ip->frag_off & 0xE000)
>> 13);
    printf("  Tiempo de vida: %d\n", encabezado_ip->ttl);
```

```
    printf("  Protocolo: %d\n", encabezado_ip->protocol);
    printf("  Suma de control: 0x%.4X\n", ntohs(encabezado_ip->check));
    printf("  Dirección IP de origen: %s\n", inet_ntoa(*(struct in_addr
*)&encabezado_ip->saddr));
    printf("  Dirección IP de destino: %s\n", inet_ntoa(*(struct
in_addr *)&encabezado_ip->daddr));

    // Encabezado TCP
    if (encabezado_ip->protocol == IPPROTO_TCP) {
        struct tcphdr *encabezado_tcp = (struct tcphdr *)(buffer +
sizeof(struct ethhdr) + encabezado_ip->ihl * 4);
        printf("Encabezado TCP\n");
        printf("  Puerto de origen: %d\n",
ntohs(encabezado_tcp->source));
        printf("  Puerto de destino: %d\n",
ntohs(encabezado_tcp->dest));
        printf("  Número de secuencia: %u\n",
ntohl(encabezado_tcp->seq));
        printf("  Número de acuse de recibo: %u\n",
ntohl(encabezado_tcp->ack_seq));
        printf("  Longitud de encabezado TCP: %d bytes\n",
encabezado_tcp->doff * 4);
        printf("  Flags: ");
        if (encabezado_tcp->syn) printf("SYN ");
        if (encabezado_tcp->ack) printf("ACK ");
        if (encabezado_tcp->fin) printf("FIN ");
        if (encabezado_tcp->rst) printf("RST ");
        if (encabezado_tcp->psh) printf("PSH ");
        if (encabezado_tcp->urg) printf("URG ");
        printf("\n");
        printf("  Tamaño de ventana: %d\n",
ntohs(encabezado_tcp->window));
        printf("  Suma de control: 0x%.4X\n",
ntohs(encabezado_tcp->check));
        printf("  Puntero de urgencia: %d\n",
ntohs(encabezado_tcp->urg_ptr));
    }

    // Encabezado UDP
    if (encabezado_ip->protocol == IPPROTO_UDP) {
        struct udphdr *encabezado_udp = (struct udphdr *)(buffer +
sizeof(struct ethhdr) + encabezado_ip->ihl * 4);
        printf("Encabezado UDP\n");
```

```
        printf("  Puerto de origen: %d\n",
ntohs(encabezado_udp->source));
        printf("  Puerto de destino: %d\n",
ntohs(encabezado_udp->dest));
        printf("  Longitud total: %d bytes\n",
ntohs(encabezado_udp->len));
        printf("  Suma de control: 0x%.4X\n",
ntohs(encabezado_udp->check));
    }

    printf("Datos:\n");
    int i;
    for (i = sizeof(struct ethhdr) + encabezado_ip->ihl * 4; i < size;
++i) {
        printf("%.2X ", buffer[i]);
        if ((i + 1) % 16 == 0) printf("\n");
    }
    printf("\n");
}

int main() {
    int sockfd;
    unsigned char buffer[BUFFER_SIZE];

    // Crear un socket raw
    if ((sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Configurar para recibir desde localhost (loopback)
    struct sockaddr_ll addr; // Corregir aquí
    socklen_t addr_len = sizeof(addr); // Corregir aquí
    addr.sll_family = AF_PACKET;
    addr.sll_protocol = htons(ETH_P_ALL);
    addr.sll_ifindex = if_nametoindex("lo");

    // Enlazar el socket a la interfaz loopback
    if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("bind");
        exit(EXIT_FAILURE);
    }
```

```
    // Recibir paquetes
    while (1) {
        int bytes_recibidos = recv(sockfd, buffer, sizeof(buffer), 0);
        if (bytes_recibidos < 0) {
            perror("recv");
            exit(EXIT_FAILURE);
        }

        procesar_paquete(buffer, bytes_recibidos);
    }

    return 0;
}
```

raw_socket_sniffer_conIP.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>

#define BUFFER_SIZE 65536

void procesar_paquete(unsigned char *buffer, int size) {
    struct iphdr *encabezado_ip = (struct iphdr *)buffer;
    unsigned short longitud_encabezado_ip = encabezado_ip->ihl * 4;

    struct sockaddr_in source, dest;
    memset(&source, 0, sizeof(source));
    memset(&dest, 0, sizeof(dest));
    source.sin_addr.s_addr = encabezado_ip->saddr;
    dest.sin_addr.s_addr = encabezado_ip->daddr;

    if (encabezado_ip->protocol == IPPROTO_TCP) {
        struct tcphdr *encabezado_tcp = (struct tcphdr *)(buffer +
longitud_encabezado_ip);
        unsigned int puerto_origen = ntohs(encabezado_tcp->source);
```

```
        unsigned int puerto_destino = ntohs(encabezado_tcp->dest);

        printf("Paquete TCP - Dirección IP de origen: %s, Puerto de
origen: %u, Dirección IP de destino: %s, Puerto de destino: %u\n",
            inet_ntoa(source.sin_addr), puerto_origen,
inet_ntoa(dest.sin_addr), puerto_destino);
    } else if (encabezado_ip->protocol == IPPROTO_UDP) {
        struct udphdr *encabezado_udp = (struct udphdr *)(buffer +
longitud_encabezado_ip);
        unsigned int puerto_origen = ntohs(encabezado_udp->source);
        unsigned int puerto_destino = ntohs(encabezado_udp->dest);

        printf("Paquete UDP - Dirección IP de origen: %s, Puerto de
origen: %u, Dirección IP de destino: %s, Puerto de destino: %u\n",
            inet_ntoa(source.sin_addr), puerto_origen,
inet_ntoa(dest.sin_addr), puerto_destino);
    } else if (encabezado_ip->protocol == IPPROTO_ICMP) {
        printf("Paquete ICMP - Dirección IP de origen: %s, Dirección IP
de destino: %s\n",
            inet_ntoa(source.sin_addr), inet_ntoa(dest.sin_addr));
    } else {
        printf("Paquete de protocolo desconocido\n");
    }
}

int main() {
    int sockfd;
    unsigned char buffer[BUFFER_SIZE];

    // Crear un socket raw
    if ((sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_TCP)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Recibir paquetes
    while (1) {
        int bytes_recibidos = recvfrom(sockfd, buffer, sizeof(buffer),
0, NULL, NULL);
        if (bytes_recibidos < 0) {
            perror("recvfrom");
            exit(EXIT_FAILURE);
        }
```

```
        procesar_paquete(buffer, bytes_recibidos);
    }

    return 0;
}
```

raw_socket_sniffer_conMAC.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <net/ethernet.h>

#define BUFFER_SIZE 65536

void procesar_paquete(unsigned char *buffer, int size) {
    struct ethhdr *encabezado_eth = (struct ethhdr *)buffer;

    // Verificar si el paquete es Ethernet
    if (ntohs(encabezado_eth->h_proto) == ETH_P_IP) {
        struct iphdr *encabezado_ip = (struct iphdr *)(buffer +
sizeof(struct ethhdr));
        unsigned short longitud_encabezado_ip = encabezado_ip->ihl * 4;

        printf("Paquete IP - Longitud: %d bytes\n", size);

        struct in_addr ip_origen, ip_destino;
        ip_origen.s_addr = encabezado_ip->saddr;
        ip_destino.s_addr = encabezado_ip->daddr;

        printf("  Dirección IP de origen: %s\n", inet_ntoa(ip_origen));
        printf("  Dirección IP de destino: %s\n",
inet_ntoa(ip_destino));

        // Mostrar direcciones MAC de origen y destino
```

```
        printf("  Dirección MAC de origen:
%.2X:%.2X:%.2X:%.2X:%.2X:%.2X\n",
               encabezado_eth->h_source[0],
encabezado_eth->h_source[1], encabezado_eth->h_source[2],
               encabezado_eth->h_source[3],
encabezado_eth->h_source[4], encabezado_eth->h_source[5]);

        printf("  Dirección MAC de destino:
%.2X:%.2X:%.2X:%.2X:%.2X:%.2X\n",
               encabezado_eth->h_dest[0], encabezado_eth->h_dest[1],
encabezado_eth->h_dest[2],
               encabezado_eth->h_dest[3], encabezado_eth->h_dest[4],
encabezado_eth->h_dest[5]);

        // Desmenuzar el protocolo
        if (encabezado_ip->protocol == IPPROTO_TCP) {
            struct tcphdr *encabezado_tcp = (struct tcphdr *)(buffer +
sizeof(struct ethhdr) + longitud_encabezado_ip);
            printf("  Protocolo: TCP\n");
            printf("  Puerto de origen: %u\n",
ntohs(encabezado_tcp->source));
            printf("  Puerto de destino: %u\n",
ntohs(encabezado_tcp->dest));
        } else if (encabezado_ip->protocol == IPPROTO_UDP) {
            struct udphdr *encabezado_udp = (struct udphdr *)(buffer +
sizeof(struct ethhdr) + longitud_encabezado_ip);
            printf("  Protocolo: UDP\n");
            printf("  Puerto de origen: %u\n",
ntohs(encabezado_udp->source));
            printf("  Puerto de destino: %u\n",
ntohs(encabezado_udp->dest));
        } else if (encabezado_ip->protocol == IPPROTO_ICMP) {
            printf("  Protocolo: ICMP\n");
        } else {
            printf("  Protocolo desconocido\n");
        }
    } else {
        printf("Paquete no Ethernet\n");
    }
}

int main() {
    int sockfd;
```

```
    unsigned char buffer[BUFFER_SIZE];

    // Crear un socket raw
    if ((sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Recibir paquetes
    while (1) {
        int bytes_recibidos = recv(sockfd, buffer, sizeof(buffer), 0);
        if (bytes_recibidos < 0) {
            perror("recv");
            exit(EXIT_FAILURE);
        }

        procesar_paquete(buffer, bytes_recibidos);
    }

    return 0;
}
```

raw_socket_sniffer.c:

```
/*
Protocolos de Internet- Javier Ouret
RAW SOCKETS VERSION SIMPLIFICADA
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>

#define BUFFER_SIZE 65536

void procesar_paquete(unsigned char *buffer, int size) {
```

```
    struct iphdr *encabezado_ip = (struct iphdr *)buffer;
    unsigned short longitud_encabezado_ip = encabezado_ip->ihl * 4;

    if (encabezado_ip->protocol == IPPROTO_TCP) {
        struct tcphdr *encabezado_tcp = (struct tcphdr *)(buffer +
longitud_encabezado_ip);
        unsigned int puerto_origen = ntohs(encabezado_tcp->source);
        unsigned int puerto_destino = ntohs(encabezado_tcp->dest);

        printf("Paquete TCP - Puerto de origen: %u, Puerto de destino:
%u\n", puerto_origen, puerto_destino);
    } else if (encabezado_ip->protocol == IPPROTO_UDP) {
        struct udphdr *encabezado_udp = (struct udphdr *)(buffer +
longitud_encabezado_ip);
        unsigned int puerto_origen = ntohs(encabezado_udp->source);
        unsigned int puerto_destino = ntohs(encabezado_udp->dest);

        printf("Paquete UDP - Puerto de origen: %u, Puerto de destino:
%u\n", puerto_origen, puerto_destino);
    } else if (encabezado_ip->protocol == IPPROTO_ICMP) {
        printf("Paquete ICMP\n");
    } else {
        printf("Paquete de protocolo desconocido\n");
    }
}

int main() {
    int sockfd;
    unsigned char buffer[BUFFER_SIZE];

    // Crear socket raw
    if ((sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_TCP)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Recibir paquetes
    while (1) {
        int bytes_recibidos = recvfrom(sockfd, buffer, sizeof(buffer),
0, NULL, NULL);
        if (bytes_recibidos < 0) {
            perror("recvfrom");
            exit(EXIT_FAILURE);
```

```
        }

        procesar_paquete(buffer, bytes_recibidos);
    }

    return 0;
}
```

4. **Con PYCHARM probar el ejemplo de cliente servidor en PYTHON.**

En el GITHUB habian 5 archivos .py para probar con Pycharm:

- Socket_Cliente_B.py
- Socket_Cliente_Select.py
- Socket_Cliente_Select_Lento.py
- Socket_Servidor_Concurrente_01.py
- Socket_Servidor_Select.py

Estableciendo conexión entre el Cliente B y el Socket Servidor Concurrente 01, sale al hacer Run lo siguiente:
Socket_Servidor_Concurrente_01:
Socket Creado
socket bind Completado
socket en modo escucha - pasivo
conexion con ('127.0.0.1', 64456).
recibido "%s" b'1234567890'
enviando mensaje de vuelta al cliente
recibido "%s" b'1234567890'
enviando mensaje de vuelta al cliente
recibido "%s" b'1234567890'
enviando mensaje de vuelta al cliente
recibido "%s" b'1234567890'
enviando mensaje de vuelta al cliente
recibido "%s" b'1234567890'
enviando mensaje de vuelta al cliente
recibido "%s" b'1234567890'
enviando mensaje de vuelta al cliente
recibido "%s" b'1234567890'
enviando mensaje de vuelta al cliente
recibido "%s" b'1234567890'
enviando mensaje de vuelta al cliente
recibido "%s" b'1234567890'
enviando mensaje de vuelta al cliente

Socket_Cliente_B:
conectando a %s puerto %s ('localhost', 6667)
Paso: 0
Servidor:
enviando "%rb" b'12345678901234567890123456789O'

recibiendo "%s" b'Conectado '
recibiendo "%s" b'con client'
recibiendo "%s" b'e'
recibiendo "%s" b'1234567890'
Paso: 1
Paso: 1
1234567890
enviando "%rb" b'12345678901234567890123456789O'
recibiendo "%s" b'1234567890'
recibiendo "%s" b'1234567890'
recibiendo "%s" b'1234567890'
Paso: 2
Paso: 2
1234567890
enviando "%rb" b'12345678901234567890123456789O'
recibiendo "%s" b'1234567890'
recibiendo "%s" b'1234567890'
recibiendo "%s" b'1234567890'
Paso: 3
Paso: 3

Estableciendo conexión entre el Cliente Select y el Socket Servidor Select, sale al hacer Run lo siguiente:

Socket_Servidor_Select:
iniciando en localhost port 10000
esperando el próximo evento
  conexión desde: ('127.0.0.1', 62464)
esperando el próximo evento
  conexión desde: ('127.0.0.1', 62465)
esperando el próximo evento
  recibido b'Este mensaje ' desde ('127.0.0.1', 62464)
esperando el próximo evento
  recibido b'Este mensaje ' desde ('127.0.0.1', 62465)
 enviando b'Este mensaje ' a ('127.0.0.1', 62464)
esperando el próximo evento
   ('127.0.0.1', 62464) cola vacía
 enviando b'Este mensaje ' a ('127.0.0.1', 62465)
esperando el próximo evento
   ('127.0.0.1', 62465) cola vacía
esperando el próximo evento
  recibido b'es enviado ' desde ('127.0.0.1', 62464)
esperando el próximo evento
 enviando b'es enviado ' a ('127.0.0.1', 62464)
esperando el próximo evento
  recibido b'es enviado ' desde ('127.0.0.1', 62465)
   ('127.0.0.1', 62464) cola vacía

esperando el próximo evento
 enviando b'es enviado ' a ('127.0.0.1', 62465)
esperando el próximo evento
   ('127.0.0.1', 62465) cola vacía
esperando el próximo evento
  recibido b'en partes.' desde ('127.0.0.1', 62464)
esperando el próximo evento
 enviando b'en partes.' a ('127.0.0.1', 62464)
esperando el próximo evento
  recibido b'en partes.' desde ('127.0.0.1', 62465)
   ('127.0.0.1', 62464) cola vacía
esperando el próximo evento
 enviando b'en partes.' a ('127.0.0.1', 62465)
esperando el próximo evento
   ('127.0.0.1', 62465) cola vacía
esperando el próximo evento
  cerrando... ('127.0.0.1', 62465)
esperando el próximo evento
  cerrando... ('127.0.0.1', 62465)
esperando el próximo evento

Socket_Cliente_Select:
conectando a localhost puerto 10000
('127.0.0.1', 62464): enviando b'Este mensaje '
('127.0.0.1', 62465): enviando b'Este mensaje '
('127.0.0.1', 62464): recibido b'Este mensaje '
('127.0.0.1', 62465): recibido b'Este mensaje '
('127.0.0.1', 62464): enviando b'es enviado '
('127.0.0.1', 62465): enviando b'es enviado '
('127.0.0.1', 62464): recibido b'es enviado '
('127.0.0.1', 62465): recibido b'es enviado '
('127.0.0.1', 62464): enviando b'en partes.'
('127.0.0.1', 62465): enviando b'en partes.'
('127.0.0.1', 62464): recibido b'en partes.'
('127.0.0.1', 62465): recibido b'en partes.'

Estableciendo conexión entre el Cliente Select Lento y el Socket Servidor Select, sale al hacer Run lo siguiente:

Socket_Servidor_Select:
iniciando en localhost port 10000
esperando el próximo evento
  conexión desde: ('127.0.0.1', 64554)
esperando el próximo evento
  recibido b'Parte 1 del mensaje' desde ('127.0.0.1', 64554)
esperando el próximo evento
 enviando b'Parte 1 del mensaje' a ('127.0.0.1', 64554)
esperando el próximo evento

('127.0.0.1', 64554) cola vacía
esperando el próximo evento
recibido b'Parte 2 del mensaje' desde ('127.0.0.1', 64554)
esperando el próximo evento
enviando b'Parte 2 del mensaje' a ('127.0.0.1', 64554)
esperando el próximo evento
('127.0.0.1', 64554) cola vacía
esperando el próximo evento
cerrando... ('127.0.0.1', 64554)
esperando el próximo evento

Socket_Cliente_Select_Lento:
conectando a localhost puerto 10000
enviando b'Parte 1 del mensaje'
enviando b'Parte 2 del mensaje'
recibidos b'Parte 1 del mens'
recibidos b'ajeParte 2 del m'
recibidos b'ensaje'
cerrando socket