Práctica 2: Comunicación entre procesos remotos mediante socket

Asignatura

Carlos Prados Sesmero 70830147z
Diego Sánchez Marcos – 70892801q

Índice

Objetivo

El objetivo de esta práctica consistirá en obtener una visión clara de las características y posibilidades de la comunicación entre procesos remotos basada en el modelo cliente/servidor, utilizando los servicios del nivel de transporte de la arquitectura TCP/IP, mediante la programación de aplicaciones clientes basadas en sockets y la utilización de fuentes bibliográficas para completar la información recibida.

Planteamiento

Los mecanismos tradicionales de comunicación entre procesos en UNIX (semáforos, colas de mensajes, memoria compartida y "pipes") permiten que procesos independientes cooperen, se excluyan mutuamente de secciones críticas e intercambien datos. A pesar de su utilidad para comunicar procesos que residen en la misma máquina, su extensión a un entorno distribuido es compleja e introduce una sintaxis diferente a la que se utiliza en otras operaciones habituales como p.e. el manejo de ficheros. La utilización de sockets permite superar estas limitaciones proporcionando una forma homogénea de enfocar la comunicación entre procesos, tanto en una máquina aislada como en un entorno de red.

Un **socket** es una abstracción que representa el extremo de una vía de comunicación lógica, y que se plantea como una generalización de los mecanismos de acceso a ficheros de UNIX. Los sockets se crean en un determinado "dominio de comunicación" (ver el fichero de cabecera socket.h) que puede ser "UNIX" en el caso de procesos residentes en el mismo sistema, o uno de los disponibles para sistemas conectados a una red (internet, Appletalk, DECnet, etc.). En BSD los sockets se utilizan mediante llamadas al sistema, mientras que en otros entornos están disponibles bajo la forma de funciones de biblioteca.

En el dominio internet (TCP/IP) los sockets permiten acceder a los servicios del nivel de transporte tanto a través de conexiones (TCP) como en modo sin conexión (UDP). La forma habitual de implantar una aplicación basada en sockets es utilizar el modelo cliente-servidor. La comunicación es asimétrica, con un cliente que solicita servicios y un servidor que los proporciona bajo demanda. Ambos procesos (cliente y servidor) se ejecutan de forma asíncrona, siendo los instantes de comunicación los puntos de sincronización en la evolución de ambos.

Desde el punto de vista de un cliente la comunicación (en el caso de utilizar un protocolo de transporte orientado a la conexión, como TCP) se realiza en cuatro etapas:

- Creación del socket local: mediante la llamada a la función socket()
- Establecimiento de la conexión: la función utilizada es connect()
- Intercambio de datos: mediante read() y write() a través del socket creado
- Liberación de la conexión: utilizando close()

La sintaxis, opciones y utilización de estas funciones se encuentra documentada en las correspondientes páginas del manual on-line del sistema.

Estudio previo

Recordemos que la arquitectura más difundida en la actualidad a nivel mundial es la arquitectura TCP/IP, la cual tiene sus orígenes en el Departamento de Defensa de los EEUU. Dicha arquitectura solo contempla 4 niveles funcionales, los cuales son los siguientes:

- **Nivel de infraestructura de red**: Soporta el diálogo con la entidad IP local.
- **Nivel de interconexión de redes**: Contempla las funciones necesarias para poder encaminar y entregar en la red destino los paquetes de datos generados desde un equipo conectado a una red origen (Protocolo IP).
- **Nivel de transporte de datos**: Incluye dos protocolos complementarios: TCP (a la cual entraremos más en detalle) y UDP.
- **Nivel de aplicación**: Proporciona el interfaz con el usuario y ofrece los servicios de alto nivel de la arquitectura, como van a ser en esta práctica, una transferencia de datos.

Como ya sabemos, el protocolo IP (Internet Protocol) es el responsable de determinar la ruta para la transferencia de datagramas entre dos redes distintas (un datagrama IP será la PDU del protocolo IP).

Dentro del nivel de transporte de datos distinguíamos dos protocolos:

- **UDP (User Datagram Protocol):** Permite añadir a los datos de una entidad de aplicación una cabecera con los identificadores de los puertos origen y destino (número entero positivo).
- **TCP (Transmisssion Control Protocol):** Proporciona un servicio fiable a nivel de transporte utilizando conexiones extremo a extremo.

Es decir, las conexiones se identifican por medio de un par de puntos extremos. Un punto extremo es un par de números enteros formado por (host, puerto), en donde host es la dirección IP de un host y puerto es el número de un puerto TCP en dicho host.

Sockets

Una forma de conseguir que dos programas se transmitan datos, basada en el protocolo TCP/IP, es la programación de sockets. Un socket no es más que un "canal de comunicación" entre dos programas que están siendo ejecutados en máquinas distintas o en la misma máquina. Evidentemente será de más utilidad realizar una programación basada en sockets cuando queramos comunicar programas en máquinas distintas, ya que para comunicar procesos diferentes en una misma máquina podremos utilizar los mecanismos tradicionales de comunicación entre procesos (semáforos, colas de mensajes, memoria compartida y “pipes”).

De un modo muy simple, se puede decir que un socket es un amanera de hablar con otra máquina, es decir, una manera de hablar con otras máquinas usando descriptores de ficheros estándar de Unix. Es decir, en Unix, todas las acciones de entrada y salida son desempeñadas escribiendo o leyendo en uno de estos descriptores de fichero, los cuales son simplemente un número entero, asociado a un fichero abierto que puede ser una conexión de red, un terminal, o cualquier otra cosa.

Desde el punto de vista de programación, un socket no es más que un "fichero" que se abre de una manera especial. Una vez abierto se pueden escribir y leer datos de él con las habituales funciones de “read()” y “write()” del lenguaje C. Entraremos más en detalle más adelante, destacando diferentes aspectos de la programación de los sockets.

Existen básicamente dos tipos de "canales de comunicación" o sockets, los orientados a conexión y los no orientados a conexión.

En el caso de un socket orientado a la conexión, los llamados **Sockets de Flujo**, ambos programas deben conectarse entre ellos mediante la utilización de socket y hasta que no esté establecida correctamente la conexión, ninguno de los dos puede transmitir datos. Esta es la parte TCP del protocolo TCP/IP, y garantiza que todos los datos van a llegar de un programa al otro correctamente. Se utiliza cuando la información a transmitir es importante, no se puede perder ningún dato y no importa que los programas se queden "bloqueados" esperando o transmitiendo datos. Si uno de los programas está atareado en otra cosa y no atiende la comunicación, el otro quedará bloqueado hasta que el primero lea o escriba los datos.

En el caso de un socket no orientado a la conexión, los llamados **Socket de Datagramas**, no es necesario que los programas se conecten. Cualquiera de ellos puede transmitir datos en cualquier momento, independientemente de que el otro programa esté "escuchando" o no. Es el llamado protocolo UDP, y garantiza que los datos que lleguen son correctos, pero no garantiza que lleguen todos. Se utiliza cuando es muy importante que el programa no se quede bloqueado y no importa que se pierdan datos.

En resumen, un socket orientado a la conexión garantiza la correcta comunicación entre dos procesos de máquinas distintas o iguales, o mejor dicho, se garantiza que no se perderá ningún dato en el transcurso de la comunicación entre ambos programas.

Un ejemplo muy básico de Sockets de Flujo es el siguiente: Enviaremos por el socket de flujo tres objetos “A, B, C”, los cuales llegarán al destino en el mismo orden.

Un ejemplo para entender claramente el protocolo UDP será el siguiente: Imaginemos que realizamos un programa que está controlando la temperatura de un horno industrial encargado de la fundición de metales y envía dicha temperatura a un ordenador central en una sala de control para que éste presente unos gráficos de temperatura y de esta manera comprobar su correcto funcionamiento. Obviamente es más importante el control del horno que la perfección de la gráfica. El programa no se puede quedar bloqueado sin atender al horno simplemente porque el ordenador que muestra los gráficos esté ocupado en otra cosa. En este caso deberemos utilizar sockets tipo UDP.

En esta práctica utilizaremos Sockets de Flujo, es decir, accederemos a los dos procesos servidor disponibles mediante conexiones TCP. Posteriormente explicaremos cuales son estos procesos.

Resumen de los conceptos básicos sobre sockets

¿**Qué son** los sockets? Los sockets son mecanismos de comunicación entre procesos que permiten que dos procesos intercambien información aunque residan en máquinas distintas.

¿**Qué proporcionan** los sockets? Proporcionan un interfaz de programación de aplicaciones (API) para la familia de protocolos de Internet TCP/IP.

¿**Qué nos permiten realizar** los sockets? Desde el punto de vista de la arquitectura de una red permiten implementar un interfaz con el nivel de transporte (TLI: Transport Layer Interface), usando los servicios de los protocolos TCP o UDP, anteriormente explicados.

¿**Cómo definiéremos** un socket? Un socket quedará definido mediante un par de direcciones IP (local y remota), un protocolo de transporte y un par de números de puerto (local y remoto).

Desarrollo de la práctica

En primer lugar, vamos a describir cual es nuestro sistema y que queremos conseguir sobre él. En el host 157.88.201.98 (gredos.eii.uva.es) se encuentran disponibles dos procesos servidores accesibles mediante conexiones TCP.

El primero es un servidor de eco que acepta una petición de conexión de un cliente y queda a la espera de que este le envíe una cadena que caracteres, que el servidor devuelve al cliente. La conexión se establece con el puerto 7500.

El segundo servidor (srobot, también basado en TCP, a través del puerto 8500) acepta una petición de conexión y responde devolviendo una estructura de datos con diferentes campos.

Es decir, vamos a tener una única máquina, con dirección IP 157.88.201.98 (Dirección clase B, con 14 bits para la red y 16 para el Host), que aplicándole una máscara de clase B (255.255.0.0) obtenemos que la parte de red será 157.88.0.0 y la parte del Host será 0.0.201.98.

Dentro de esa máquina, al igual que en el resto, existe un grupo de puntos abstractos de destino, llamados puertos (número entero positivo). Para conseguir comunicarnos con un proceso remoto, como puede ser el host 157.88.201.98 necesitaremos conocer no solo la dirección IP de la máquina, sino también el número de puerto que está utilizando el destinatario.

Es decir, cuando queramos acceder a la información del primer servidor (eco) lo haremos con el puerto 7500, mientras que para acceder al segundo servidor lo haremos a través del puerto 8500). Cabe destacar que los 1024 primeros puertos están oficialmente asignados a servicios concretos, por esta razón los puertos asignados a los dos servicios anteriores tienen un valor tan elevado.

1. Cliente eco

En este apartado vamos a completar el cliente de eco para que funcione correctamente el código aportado en el enunciado de la práctica. De tal manera que nos queda el siguiente código:

```c
//Cliente de eco sobre el protocolo TCP.

#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<stdlib.h>
#include<errno.h>
#include<sys/unistd.h>
#include<netdb.h>

#define SERV_TCP_PORT 7500 // Puerto que utiliza el servidor para eco
#define SERV_HOST_ADDR "157.88.201.95"
#define MAXLINE 512

main(int argc, char *argv[])
{
    //Variables de programa

    int sockfd, sockConect, sockWrite, sockRead, cerrarSocket;
    struct sockaddr_in serv_addr;
    char DATA[MAXLINE];
    char Recibido[MAXLINE];

    //Ponemos a 0 la estructura de direcciones remotas
    bzero((char *) &serv_addr, sizeof(serv_addr));

    //Rellenamos la estructura de direcciones remotas
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port = htons(SERV_TCP_PORT);

    /*Creamos un socket para que el cliente pueda enviar y recibir mensajes
    AF_INET para sockets IPv4, SOCK_STREAM para sockets TCP y 0 porque es una aplicacion típica*/
    sockfd=socket(AF_INET,SOCK_STREAM,0);
    //Comprobamos errores
    if (sockfd<0)
    {
        perror("client: can't open stream socket");
        exit(EXIT_FAILURE);
    }

    //Conectamos el socket con el servidor
    sockConect=connect(sockfd,(struct sockaddr *) &serv_addr, sizeof(serv_addr));
    //Comprobamos errores
    if (sockConect < 0)
    {
        perror("client: can't connect to server");
        exit(EXIT_FAILURE);
    }
```

```
54      //Leemos el mensaje
55      printf("\nINTRODUCIR MENSAJE: ");
56      gets(DATA);
57      printf("\n\n");
58
59      //Escribimos en el socket
60      sockWrite=write(sockfd, DATA, sizeof(DATA));
61      //Comprobamos errores
62      if (sockWrite < 0)
63      {
64          perror("writing on stream socket");
65          exit(EXIT_FAILURE);
66      }
67
68      //Leemos el eco del socket
69      sockRead=read(sockfd, Recibido, sizeof(Recibido));
70      //Comprobamos errores
71      if (sockRead < 0)
72      {
73          perror("reading stream message");
74          exit(EXIT_FAILURE);
75      }
76
77      //Imprimimos el eco recibido.
78      printf("Datos recibidos: %s\n", Recibido);
80      //Cerramos el socket
81      cerrarSocket=close(sockfd);
82      //Comprobamos errores
83      if(cerrarSocket<0)
84      {
85          perror("close()");
86          exit(EXIT_FAILURE);
87      }
88
89      exit(0);
90  }
```

Como podemos observar definiremos la dirección IP y el puerto nombrados anteriormente para el servidor de eco, con un máximo de mensaje de 512.

- Inicialización de variables:
  - Nos apoyaremos en las variables tipo “int” llamadas: “sockfd”, “sockConect”, “sockWrite”, “sockRead” y “cerrarSocket”. Estas variables las hemos utilizado para realizar un código más sencillo a la hora realizar las tareas de crear, enlacar, etc. los sockets. Nos servirán también para la comprobación de errores posteriores a la realización de dichas tareas.
  - “serv_addr”: Inicializaremos esta variable con una estructura “sockaddr_in” que posteriormente explicaremos. Básicamente el objetivo que se propone a esta variable será contener las direcciones del servidor, tanto la dirección IP como el puerto. En un apartado posterior de esta práctica se explicarán los campos y función de dicha estructura.
  - “DATA” y “Recibido”: Serán cadenas de caracteres con un tamaño máximo de MAXLINE (512). Serán las encargadas de almacenar, respectivamente, la cadena de caracteres que introducimos y enviamos al servidor a través del socket y la cadena de caracteres recibida procedente del servidor a través del socket. Hemos decidido crear dos cadenas de caracteres en lugar de una para garantizar su funcionamiento, ya que si sobrescribimos una única cadena los datos mostrados podrían ser los datos escritos en la máquina cliente.

- Creación del socket cliente: Para ello deberemos realizar una llamada a la función socket(). Almacenaremos el valor que nos devuelve dicha función en la variable llamada "sockfd" para su posterior comprobación de errores.

  Previo a crear el socket deberemos rellenar la estructura de direcciones remotas, es decir, definir las direcciones del servidor. Una vez hecho esto llamaremos a la primitiva crearSocket (función "socket()"). Aunque este punto no será crítico para la creación del socket.

  Para la creación de dicho socket hemos especificado tres campos (los cuales explicaremos más adelante). Estos campos indican que el socket tiene las siguientes características:

  - Utilizará un entorno de comunicaciones de la familia de protocolos IPv4 (al igual que hemos definido la variable que almacena las direcciones de la máquina servidor).
  - Será un socket basado en conexiones, es decir, un socket TCP o de flujo de datos, garantizando de esta manera la comunicación.
  - El socket tomará un protocolo por defecto basándose en las dos características anteriores, en este caso el protocolo será TCP.

  Explicaremos más al detalle está función más adelante en esta misma práctica.

  Posteriormente comprobaremos si ha habido algún error, en cuyo caso se mandará un mensaje de error al usuario por pantalla.

- Conexión con el servidor: Para ello realizaremos una llamada a la función "connect()". En este caso será necesario haber definido la estructura de las direcciones del servidor, en caso contrario no será posible la comunicación.

  Para enlazar dicho socket al servidor hemos especificado tres campos (los cuales explicaremos más adelante detalladamente):

  - Estableceremos que la conexión será con el socket almacenado en la variable "sockfd" (número entero que identifica al socket).
  - Definiremos que enlazamos el socket con un servidor, por lo que introduciremos en uno de los campos la variable struct que almacena las direcciones del servidor.
  - Definiremos el tamaño de la variable que almacena las direcciones del servidor.

  Posteriormente comprobaremos si ha habido algún error, en cuyo caso se mandará un mensaje de error al usuario por pantalla.

- Envío de datos al servidor: Previamente, vamos a introducir por teclado la cadena de caracteres que queremos enviar al servidor. Una vez hecho esto escribiremos en el socket a través de la siguiente instrucción:

  sockWrite=write(sockfd, DATA, sizeof(DATA));

  write() escribe hasta "sizeof(DATA)" bytes de datos al socket especificado. Donde los argumentos de dicha función especial son los siguientes:

- Fd (descriptor del socket creado con anterioridad): en nuestro caso “sockfd”.
- Buf (Buffer que contiene los datos a escribir): En nuestro caso será la cadena de caracteres solicitada anteriormente y almacenada en DATA.
- Num (número de bytes a escribir en el sockets): En nuestro caso será el tamaño de la cadena de caracteres DATA.

Al igual que en el resto de casos realizaremos un tratamiento de errores, cuyo código se encuentra justo después de la llamada a la función write() y que tendrá las mismas características que en los casos anteriores.

- Recepción de datos del servidor: Para ello hemos desarrollado la siguiente instrucción:

  sockRead=read(sockfd, DATA, sizeof(DATA));

  read() lee datos del socket, para lo cual tendrá los siguientes argumentos:
  - Fd (Descriptor del socket creado con anterioridad): en nuestro caso “sockfd”.
  - Buf (Buffer que contendrá los datos que se lean): Los almacenaremos en la cadena de caracteres DATA.
  - N_bytes (Longitud del buffer en bytes): Indica el tamaño máximo en bytes de los datos a leer, pues debe ser como máximo igual al tamaño de Buf. En nuestro caso podremos que será el tamaño de la cadena de caracteres DATA.

  Cabe destacar que la función read() bloqueará el programa hasta que haya algo para que se lea en el socket, por lo que en muchos casos deberemos garantizar la correcta programación del servidor para evitar bloqueos.

El resultado que hemos obtenido al ejecutar nuestro programa cliente de eco es el siguiente:

```
[cci48@carpanta P2]$ ./cliente
INTRODUCIR MENSAJE: Este mensaje sera devuelto por el servidor de eco
Datos recibidos: Este mensaje sera devuelto por el servidor de eco
```

2. Cabecera <sys/socket.h>, <netlinet/in.h> y <arpa/inet.h>

sys/socket.h

Es la cabecera principal de los sockets, contiene definiciones sobre los sockets. Este archivo de cabecera incluye varias definiciones de estructuras necesarias para los sockets. La cabecera <sys/socket.h> definirá los siguientes tipos de datos:

- socklen_t:
- sa_family_t
- sockaddr: Usada para definir la dirección de un socket que es usada en: bind(), connect(), getpeername(), etc.

El archivo de cabecera <sys/socket.h> tiene definidas diferentes estructuras que introduciremos a la hora de programar mediante sockets, estas serán las siguientes:

- struct sockaddr:

```
1. struct sockaddr
2. {
3.     unsigned short integer sa_family;     // address family
4.     char sa_data[14];      //  up to 14 bytes of direct address
5. };
```

Utilizada en nuestro programa de cliente de eco a la hora de enlazar el socket a una dirección IP y a un puerto local y a la hora de conectar el socket con el servidor.

- struct in_addr:

```
1. struct in_addr
2. {
3.         unsigned long integer s_addr;
4. };
```

Se utilizaría para definir una estructura de una dirección IP.

- struct sockaddr_in

```
1. struct sockaddr_in
2. {
3.         short   sin_family;
4.         unsigned short integer sin_port;
5.         struct  in_addr sin_addr;
6.         char    sin_zero[8];
7. };
```

Se utiliza para definir una estructura de la dirección del servidor y del cliente, tendrá diferentes campos que nos ayudarán a definir correctamente la dirección del cliente y del servidor. En nuestro programa la hemos utilizado para definir la estructura de direcciones remotas (veáse apartado 1). La cabecera <sys/socket.h> nos ayudará a utilizar dicha estructura, aunque será la siguiente cabecera la que la almacena.

Netinet/in.h

Este archivo de cabecera contiene constantes y estructuras necesarias para el dominio de direcciones de Internet o direcciones IP. Contiene definiciones para la familia de protocolos de Internet. Cuando introducimos el archivo de cabecera <netinet/in.h>, definimos, mediante typedef, los siguientes tipos de datos:

- "**in_port_t**": Será una variable de tipo entero sin signo (unsigned int) de 16 bits. Definirá el puerto.
- "**in_addr_t**": Será otro unsigned int de 32 bits. Definirá la dirección IP de una máquina.

Esta cabecera también define la estructura "in_addr", la cual incluye al menos el siguiente miembro:

in_addr_t s_addr

La cabecera incluye una estructura llamada "sockaddr_in" la cual es muy utilizada en las aplicaciones donde nos basamos en el uso de sockets. Esta estructura la definiremos en un apartado posterior de la práctica, pero adelantamos que esta estructura se utilizará para definir variables donde almacenaremos direcciones para la familia de protocolos de Internet en las que se incluyen el puerto y la dirección IP. Por esta razón será necesario incluir la cabecera <netinet/in.h> cuando realicemos comunicación mediante sockets (ya que en caso contrario no podremos establecer direcciones).

Arpa/inet.h

La cabecera <arpa/inet.h> pone a disposición el tipo "in_port_t", el tipo "in_addr_" y el tipo "in_addr" tal y como se define en la descripción de la cabecera <netinet/in.h>. Es decir, contiene definiciones para las operaciones de Internet.

Entre las funciones que incluye este fichero de cabecera encontramos las funciones:

- Htonl
- Htons
- Ntohl
- Ntohs

Estas funciones las utilizaremos para garantizar la correcta comunicación entre máquinas, garantizando el correcto orden de los bytes dentro de un conjunto de bytes. Explicaremos esto detalladamente más adelante.

3. "struct sockaddr_in"

El direccionamiento de clientes y servidores mediante sockets se realiza utilizando la información contenida en variables del tipo "struct sockaddr_in".

Esta estructura sirve para manejar direcciones de internet (IP). Es una estructura básica para todas las llamadas "sys" y funciones que se ocupan de las direcciones IP. En memoria, esta estructura es del mismo tamaño que la estructura "struct sockaddr", pudiendo asignar libremente el puntero de un tipo de estructura al otro sin ningún tipo de problema. Queda definida de la siguiente manera:

```
#include <netinet/in.h>

struct sockaddr_in
{
    short            sin_family;   // e.g. AF_INET
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};
```

Estos cuatro argumentos son los siguientes:

- "sin_family": Define el entrono de comunicaciones. Puede ser en un entorno de comunicación local (AF_UNIX), un entorno más amplio que utilice la familia de protocolos IPv4, etc.
- "Sin_port": Define el puerto de la variable que definimos con la estructura "sockaddr_in".
- "Sin_addr": Define la dirección IP de la variable que definimos con la estructura "sockaddr_in". Es de tipo "struct in_addr" (definida posteriormente).
- "sin_zero[8]": Es un argumento de la estructura opcional que, de momento, nosotros no vamos a utilizar.

Esto quiere decir que para definir la dirección IP y el puerto de una máquina con la que queremos comunicarnos deberemos definir una estructura "sockaddr_in". Como podemos observar el argumento "sin_addr" de dicha estructura será un tipo de dato "struct", por lo que debemos introducir el concepto de estructura "struct in_addr":

```
struct in_addr
{
    unsigned long s_addr;  // load with inet_aton()
};
```

Esta estructura tendrá un único argumento, que será el que vemos en el código superior. En nuestra práctica hemos introducido la estructura de la dirección del servidor de la siguiente manera:

```
struct sockaddr_in serv_addr;

//Rellenamos la estructura de direcciones remotas
 serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
serv_addr.sin_port = htons(SERV_TCP_PORT);
```

En primer lugar hemos definido una variable "serv_addr" que utiliza la estructura de la que estamos hablando. Posteriormente rellenamos la estructura de direcciones remotas (definimos la estructura de la dirección del servidor). Para ello hemos definido:

- "sin_familia" será AF_INET, es decir, los siguientes datos introducidos en la estructura de direcciones remotas serán de la familia de protocolos IPv4. De esta manera aseguramos que utilizamos un protocolo TCP, ya que utilizamos un socket de flujo de datos.
- "sin_addr.s_addr": Accedemos al argumento "s_addr" de la estructura "sin_addr" para introducir una dirección IP. En el caso del programa del cliente de eco introduciremos la dirección IP del servidor, la cual está almacenada en la variable "SERV_HOST_ADDR".

- “sin_port”: Aquí definiremos el puerto a utilizar por el servidor, el cual está almacenado en la variable “SERV_TCP_PORT”.

4. Llamadas a “socket” y “connect”

1) Creación de socket cliente: Crearemos un socket para que el cliente pueda enviar y recibir mensajes procedentes del servidor. Para ello hemos escrito la siguiente instrucción:

sockfd=socket(AF_INET , SOCK_STREAM , 0);

socket() crea un nuevo socket de un determinado tipo, identificado mediante un entero, y le asigna recursos del sistema, es decir, crea un punto de acceso al canal de comunicación y devuelve un descriptor del socket. Tendrá los siguientes argumentos:

a. Domain (entorno de comunicaciones): En este caso hemos puesto un entorno “AF_INET” que nos servirá para poder establecer la comunicación a través de internet. Es decir, utilizará la familia de protocolos IPv4. Gracias a esto podremos establecer una comunicación, no solo local, sino también con máquinas con diferentes direcciones IP.
b. Type: En este argumento hemos decidido poner “SOCK_STREAM”, el cual nos aportará un servicio basado en conexión, es decir, un socket de flujo basado en el protocolo TCP/IP (no UDP). De esta manera podremos garantizar la correcta comunicación entre cliente y servidor.
c. Protocolo: Pondremos un “0” (por defecto), es decir, elegirá internamente el protocolo partiendo de los argumentos que hemos puesto en “domain” y “type”. Será una aplicación típica (0).

Posteriormente comprobaremos errores en la creación del socket e imprimiremos por pantalla el error dado con las siguientes instrucciones:

```
if (sockfd<0)
{
    perror("client: can't open stream socket");
    exit(EXIT_FAILURE);
}
```

La llamada al sistema de socket devuelve un número entero pequeño, valor que se utilizará para todas las referencias posteriores a ese socket. De tal manera que si el socket falla nos devolverá un -1, caso en el que se sale del programa.

2) Conexión con el servidor: Para ello deberemos realizar un paso previo, que será definir la estructura de direcciones del servidor. Para ello rellenamos la estructura de direcciones remotas con el siguiente código:

```
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
serv_addr.sin_port = htons(SERV_TCP_PORT);
```

Es decir, la estructura de direcciones remotas estará basada en la familia de protocolos IPv4 y tendrá la dirección IP y el puerto definidos al inicio del programa. Para conectar el socket con el servidor hemos realizado la siguiente instrucción:

*sockConect=connect(sockfd , (struct sockaddr *) &serv_addr , sizeof(serv_addr));*

connect() asigna un número libre de puerto local al socket. En el caso de un TCP socket como es nuestro caso, intenta establecer una nueva conexión TCP, es decir, conecta un socket, identificado mediante su descriptor, a una dirección remota que pasa como argumento. Tendrá los siguientes argumentos:

a. Fd (Descriptor del socket): Debería configurarse como el fichero descriptor del socket, el cuál fue devuelto por la llamada a "socket()", por lo que será un número entero positivo. Es decir, en nuestro caso será "sockfd". Será el número asignado al socket creado anteriormente.
b. Serv_addr (Dirección del host al que se desea conectar, incluyendo el puerto): Es un puntero a la estructura "sockaddr" la cuál contiene la dirección IP destino y el puerto. De tal manera que en nuestro caso será "serv_addr" con su correspondiente estructura, es decir, incluiremos todos los parámetros que quedan definidos en esa variable gracias a la estructura "sockaddr" (véase apartado 3 de esta práctica).
c. Addrlen (Tamaño): Será el tamaño de la dirección del host al que queremos conectarnos, para ello nos apoyamos en la función "sizeof".

Hay que tener en cuenta que el cliente necesita saber el número de puerto del servidor, pero no necesita saber su propio número de puesto. Esto se asignará por el sistema cuando se llama a "connect()". Para el tratamiento de errores, al igual que en el caso anterior, hemos realizado el siguiente código:

```
if (sockConect < 0)
{
    perror("client: can't connect to server");
    exit(EXIT_FAILURE);
}
```

5. Función "htons"

Las diferentes máquinas usan diferente orden a la hora de colocar los bytes internamente para sus enteros de tamaño mayor a un char. Por ejemplo, una máquina puede enviar un número de dos bytes a otra (00000001 00000000 -> 256). Dependiendo de qué byte tome como byte más significativo tendremos un correcto entendimiento entre máquinas o no. Si la máquina destino toma el número como (00000000 00000001 -> 1) tendremos una mala coordinación entre ambas máquinas.

Esto se debe a que existen dos convenios: Big Endian (BE, byte de la izquierda es el más significativo) y Little Endian (LE, byte de la derecha es el más significativo).

Este problema de transmisión quedó resuelto gracias a un convenio llamado "Network Byte Order" que define un convenio Big Endian. Pero para evitar fallos se debe hacer una conversión de BE a LE y viceversa, para lo cual utilizamos una librería de funciones:

- htons -> Host to network short
- ntohs -> Host to network long
- htonl -> Network to host short
- ntohl -> Network to host long

Esto significa la conversión del formato del host/red network al formato de la red network/host a formato small o long (16 y 32 bits respectivamente).

No necesariamente necesitamos saber en qué formato trabaja nuestra máquina, ya que se pone siempre por defecto y será el linker el encargado de detectar de que tipo es nuestra máquina (BE o LE). De manera que si introducimos una instrucción de conversión de formato y no necesitamos cambiar dicho formato, no lo cambiará.

En resumen, la función "htons" sirve para cambiar el orden de bytes que utiliza el host al orden de bytes que utiliza la red (que será BE debido al convenio acordado). Si no realizamos esta conversión en nuestro lado cliente (y si es necesaria), el servidor recibirá la información en orden inverso y no podremos garantizar la correcta comunicación entre cliente y servidor.

6. Conexión con el servidor srobot

En este apartado vamos a modificar el programa que hemos generado en el apartado 1 para que el cliente pueda conectarse al servidor **srobot**, que se encuentra disponible en el mismo host que el servidor de eco (por lo que no cambiará su dirección IP), a través del puerto TCP 8500 como habíamos descrito en el inicio de la práctica.

El servidor, al recibir una conexión de un cliente devuelve información sobre los datos del robot en una variable del tipo "struct datos_robot" cuya definición es la siguiente:

```
struct datos_robot{
    char modo;              // 'x': ningún modo activo
                            // 'm': manual sin HOME
                            // 'M' manual con HOME
                            // 'D': MDI
                            // 'A': automático
    char motor_activo;      // '0': ningún motor activo
                            // '1/2/3': activo el motor 1/2/3
                            // '4': todos los motores activos

    char tipo_interpolacion_G;      // '0', '1', '2', '3', '4'
    float vel_F;            // velocidad lineal
    float radio_giro_R;     // radio
    struct posicion pos_XY_final;   // posición final de la trayectoria actual
    struct posicion pos_XY_cmd;     // posición comandada (o interpolada)
    struct posicion pos_XY_real;    // posición real
};
```

Siendo la estructura struct posición definida de la siguiente forma:

```
struct posicion{
        float X;
        float Y;
};
```

Para ello vamos a realizar un programa, donde el cliente deberá abrir la conexión, enlazar el socket a un servidor, recibir la información, mostrarla por la pantalla y cerrar dicha conexión, como el siguiente:

```
2   //Cliente del robot sobre el protocolo TCP.
3
4   #include<stdio.h>
5   #include<sys/types.h>
6   #include<sys/socket.h>
7   #include<netinet/in.h>
8   #include<arpa/inet.h>
9   #include<stdlib.h>
10  #include<errno.h>
11  #include<sys/unistd.h>
12  #include<netdb.h>
13
14  #define SERV_TCP_PORT 8500
15  #define SERV_HOST_ADDR "157.88.201.95"
16  #define MAXLINE 512
17
18  float ReverseFloat(float);
19
20  struct posicion
21  {
22      float X;
23      float Y;
24  };
```

```c
struct datos_robot
{
    char modo;                      // 'x': ningún modo activo
                                    // 'm': manual sin HOME
                                    // 'M' manual con HOME
                                    // 'D': MDI
                                    // 'A': automático

    char motor_activo;              // '0': ningún motor activo
                                    // '1/2/3': activo el motor 1/2/3
                                    // '4': todos los motores activos

    char tipo_interpolacion_G;      // '0', '1', '2', '3', '4'
    float vel_F;                    // velocidad lineal
    float radio_giro_R;             // radio
    struct posicion pos_XY_final;   // posición final de la trayectoria actual
    struct posicion pos_XY_cmd;     // posición comandada (o interpolada)
    struct posicion pos_XY_real;    // posición real
};

main(int argc, char *argv[])
{
    //Variables de programa

    int sockbn, sockfd, sockConect, sockWrite, sockRead, cerrarSocket;
    struct sockaddr_in serv_addr;
    struct datos_robot rec;

    //Ponemos a 0 la estructura de direcciones remotas
    bzero((char *) &serv_addr, sizeof(serv_addr));

    //Rellenamos la estructura de direcciones remotas
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port = htons(SERV_TCP_PORT);

    /*Creamos un socket para que el cliente pueda enviar y recibir mensajes
    AF_INET para sockets IPv4, SOCK_STREAM para sockets TCP y 0 porque es una aplicacion típica*/
    sockfd=socket(AF_INET,SOCK_STREAM,0);
    //Comprobamos errores
    if (sockfd<0)
    {
        perror("client: can't open stream socket");
        exit(EXIT_FAILURE);
    }

    //Conectmos el socket con el servidor
    sockConect=connect(sockfd,(struct sockaddr *) &serv_addr, sizeof(serv_addr));
    //Comprobamos errores
    if (sockConect < 0)
    {
        perror("client: can't connect to server");
        exit(EXIT_FAILURE);
    }

    //Leemos los datos del socket
    sockRead=read(sockfd, &rec, sizeof(rec));
    //Comprobamos errores
    if (sockRead < 0)
    {
        perror("reading stream message");
        exit(EXIT_FAILURE);
    }
```

```
90      //Imprimimos los datos recibido.
91      printf("\nDATOS RECIBIDOS: \n\n");
92      printf("Modo de funcionamiento: ");
93      switch (rec.modo)
94      {
95          case 'x':
96              printf("Ningun modo activo.\n");
97              break;
98          case 'm':
99              printf("Manual sin HOME.\n");
100             break;
101         case 'M':
102             printf("Manual con HOME.\n");
103             break;
104         case 'D':
105             printf("MDI.\n");
106             break;
107         case 'A':
108             printf("Automatico.\n");
109             break;
110     }
112     printf("\nMotores activos: ");
113     switch (rec.motor_activo)
114     {
115         case '0':
116             printf("Ningun motor activo.\n");
117             break;
118         case '1':
119             printf("Activo el motor 1.\n");
120             break;
121         case '2':
122             printf("Activo el motor 2.\n");
123             break;
124         case '3':
125             printf("Activo el motor 3.\n");
126             break;
127         case '4':
128             printf("Todos los motores activos.\n");
129             break;
130     }
131
132     printf("\nTipo interpolacion G: %c\n", rec.tipo_interpolacion_G);
133
134     // Cambiamos el formato de los floats (LD-BE)
136     rec.vel_F=ReverseFloat(rec.vel_F); // Formato de la velocidad lineal
137     rec.radio_giro_R=ReverseFloat(rec.radio_giro_R); // Formato del radio
138     rec.pos_XY_final.X=ReverseFloat(rec.pos_XY_final.X); // Formato de la posicion final
139     rec.pos_XY_final.Y=ReverseFloat(rec.pos_XY_final.Y);
140     rec.pos_XY_cmd.X=ReverseFloat(rec.pos_XY_cmd.X); // Formato de la posicion interpolada
141     rec.pos_XY_cmd.Y=ReverseFloat(rec.pos_XY_cmd.Y);
142     rec.pos_XY_real.X=ReverseFloat(rec.pos_XY_real.X); // Formato de la posicion real
143     rec.pos_XY_real.Y=ReverseFloat(rec.pos_XY_real.Y);
144
145     // Imprimimos los floats correctamente
146
147     printf("\nVelocidad: %f\n", rec.vel_F);
148     printf("\nRadio de giro: %f\n", rec.radio_giro_R);
149     printf("\nPosicion final (X,Y): (%f,%f)\n", rec.pos_XY_final.X,rec.pos_XY_final.Y);
150     printf("\nPosicion interpolada (X,Y): (%f,%f)\n", rec.pos_XY_cmd.X,rec.pos_XY_cmd.Y);
151     printf("\nPosicion real (X,Y): (%f,%f)\n", rec.pos_XY_real.X,rec.pos_XY_real.Y);
152
153     //Cerramos el socket
154     cerrarSocket=close(sockfd);
155     //Comprobamos errores
156     if(cerrarSocket<0)
157     {
158         perror("close()");
159         exit(EXIT_FAILURE);
160     }
161
162     exit(0);
163 }
```

```
165   float ReverseFloat(float inFloat)
166   {
167       float retVal;
168       char *floatToConvert = ( char* ) & inFloat;
169       char *returnFloat = ( char* ) & retVal;
170
171       // swap the bytes into a temporary buffer
172       returnFloat[0] = floatToConvert[3];
173       returnFloat[1] = floatToConvert[2];
174       returnFloat[2] = floatToConvert[1];
175       returnFloat[3] = floatToConvert[0];
176
177       return retVal;
178   }
```

Para realizar este programa hemos tenido que hacer determinadas modificaciones sobre el programa generado en el apartado 1 de la práctica. Los detalles a destacar en este programa son los siguientes:

1. Modificar el puerto de la máquina destino, en este caso será el puerto 8500.
2. Definir las estructuras “datos_robot” y “posición”. Las cuales están desarrolladas de la misma manera en la máquina destino (servidor srobot).

3. Las variables de programa asociadas a las funciones de los sockets y la variable que define las direcciones del servidor van a permanecer invariadas. Deberemos definir una variable con una estructura “datos_robot” llamada “rec” (por ejemplo) en la que almacenaremos los datos recibidos procedentes del servidor. Podremos eliminar las variables de cadena de caracteres que teníamos en el programa cliente de eco.

4. No modificaremos la manera de rellenar la estructura de direcciones remotas (del servidor), pero se modificará el número de puerto a utilizar.

5. Tampoco modificaremos la manera de crear el socket, ni la manera de conectar el socket al servidor, ya que será necesaria la misma configuración.

6. Suprimiremos la función “write()” ya que no será necesario incluir ninguna información a enviar, simplemente cuando generamos el socket y lo enlazamos al servidor, el servidor nos enviará la información necesaria.

7. Modificaremos los parámetros de la función “read()” de tal manera que ahora no recibiremos una cadena de caracteres, sino una variable llamada “rec” (struct), donde se almacenarán los datos recibidos. Deberemos poner un “&” para acceder a la dirección de memoria de la variable donde lo vamos a almacenar.

8. Para imprimir los datos tipo “char” de la variable “rec” (struct) habrá que realizar una serie de distinciones y su posterior impresión por pantalla.

9. Para imprimir los datos de tipo “float” deberemos realizar una conversión previa de los datos, ya que no recibimos los datos de este tipo (en este caso) de una manera correcta. De esta manera debemos cambiar el formato (de Big Endian a Little Endian o viceversa) para poder observar el verdadero valor de las variables. Esto lo realizaremos a partir de

una función que hemos generado llamada “ReverseFloat()”, la cual explicaremos en siguientes puntos.

10. Imprimiremos los valores de las variables tipo “float”, una vez modificado su formato.

11. No modificaremos la manera de cerrar el socket.

12. La función generada llamada “ReverseFloat()” va a tener el siguiente objetivo: Cambiar el formato de las variables tipo “float” de manera que el byte más significativo sea ahora el byte menos significativo y viceversa, de tal forma que podamos obtener el verdadero valor de los datos enviados por el servidor. El argumento que tendrá dicha función será un valor “float” de entrada y nos devolverá como resultado un valor “float”. La manera de realizar dicha conversión es la que podemos observar en el código superior y lo realizará en base a punteros que asignan un tamaño de un byte e invierte su orden.

Para comprobar su funcionamiento lo hemos ejecutado en nuestra máquina, obteniendo el siguiente resultado:

```
[cci48@carpanta P2]$ ./robot
DATOS RECIBIDOS:
Modo de funcionamiento: Automatico.
Motores activos: Todos los motores activos.
Tipo interpolacion G: 0
Velocidad: 0.000000
Radio de giro: 0.124563
Posicion final (X,Y): (10.000000,0.000000)
Posicion interpolada (X,Y): (8.200000,0.000000)
Posicion real (X,Y): (8.100000,0.000000)
```

Observamos que son datos más que razonables por lo que asumimo el correcto funcionamiento de nuestro programa.

7. Programa usuario del cliente de eco

En este apartado vamos a realizar un programa que actúe como usuario del cliente de eco. De tal manera que el cliente de eco funcione como una entidad de aplicación y el programa que se debe diseñar es un proceso usuario del anterior. El programa gestor deberá solicitar el nombre de un fichero de texto de cualquier tamaño, leer su contenido y mandárselo al cliente de eco, transfiriendo su contenido al servidor. Cuando el servidor envíe el contenido del fichero al cliente de eco, este debe enviarlo al gestor, mostrándolo este último por pantalla.

Para realizar esto hemos decidido que el gestor envíe línea a línea el fichero (opción más simple) y el proceso de paso de información se produzca cíclicamente hasta que se acabe el

fichero. Para la comunicación entre la aplicación “gestor” y la aplicación “cliente_b” hemos utilizado colas de mensajes, de tal modo que hemos obtenido la siguiente aplicación de “gestor”:

```
// GESTOR

#include<stdio.h>
#include<sys/types.h>
#include<stdlib.h>
#include<errno.h>
#include<sys/ipc.h>
#include<sys/msg.h>

#define MAXTEXT 100

struct Mi_Tipo_Mensaje
{
 long Id_Mensaje;
 int Dato_Numerico;
 char Mensaje[MAXTEXT];
}Un_Mensaje;

main(int argc, char *argv[])
{
    //Variables de programa
    char Datos[MAXTEXT];

    char *devf;
    char fichero[MAXTEXT];

    // Solicito el nombre del fichero
    printf("\nNombre del fichero: ");
    gets(fichero);

    // Definimos archivo
    FILE *archivo;
    archivo = fopen(fichero,"r");
    *Datos="";

    // Cola de mensajes
    key_t Clave1;
    int Id_Cola_Mensajes;

    // Obtengo una clave para la cola de mensajes
    Clave1 = ftok("/bin/ls", 35);
    if (Clave1 == (key_t)-1)
    {
        printf("Error al obtener clave para cola mensajes");
        return(0);
    }

    // Se crea la cola de mensajes y se obtiene un identificador para ella.
    // El IPC_CREAT indica que cree la cola de mensajes si no lo está.
    // el 0660 son permisos de lectura y escritura para el usuario que lance
    // los procesos. Es importante el 0 delante para que se interprete en octal.
```

```
    // Solicito el nombre del fichero
    printf("\nNombre del fichero: ");
    gets(fichero);

    // Declaramos archivo
    FILE *archivo;
    archivo = fopen(fichero,"r");
    *Datos="";

    // Cola de mensajes
    key_t Clave1;
    int Id_Cola_Mensajes;

    // Obtengo una clave para la cola de mensajes
    Clave1 = ftok("/bin/ls", 35);
    if (Clave1 == (key_t)-1)
    {
        printf("Error al obtener clave para cola mensajes");
        return(0);
    }

    // Se crea la cola de mensajes y se obtiene un identificador para ella.
    // El IPC_CREAT indica que cree la cola de mensajes si no lo está.
    // el 0660 son permisos de lectura y escritura para el usuario que lance
    // los procesos. Es importante el 0 delante para que se interprete en octal.
```

```
        // Se envía el mensaje. Los parámetros son:
        // - Id de la cola de mensajes.
        // - Dirección al mensaje, convirtiéndola en puntero a (struct msgbuf *)
        // - Tamaño total de los campos de datos de nuestros mensajes
        // - Unos flags. IPC_NOWAIT indica que si el mensaje no se puede enviar
        // (habitualmente porque la cola de mensajes está llena), que no espere
        // y de un error. Si no se pone esta flag, el programa queda bloqueado
        // hasta que se pueda enviar el mensaje.

        msgsnd (Id_Cola_Mensajes, (struct msgbuf *)&Un_Mensaje,sizeof(Un_Mensaje.Dato_Numerico)+sizeof(Un_Mensaje.Mensaje),IPC_NOWAIT);

        // RECIBIR

        // Se recibe un mensaje del otro proceso. Los parámetros son:
        // - Id de la cola de mensajes.
        // - Dirección del sitio en el que queremos recibir el mensaje,
        // convirtiéndola en puntero a (struct msgbuf *).
        // - Tamaño máximo de nuestros campos de datos.
        // - Identificador del tipo de mensaje que queremos recibir. En este caso
        // se quiere un mensaje de tipo 2. Si ponemos tipo 1, se extrae el mensaje
        // que se acaba de enviar en la llamada anterior a msgsnd().
        // - flags. En este caso se quiere que el programa quede bloqueado hasta
        // que llegue un mensaje de tipo 2. Si se pone IPC_NOWAIT, se devolvería
        // un error en caso de que no haya mensaje de tipo 2 y el programa
        // continuaría ejecutándose.

        msgrcv (Id_Cola_Mensajes, (struct msgbuf *)&Un_Mensaje,sizeof(Un_Mensaje.Dato_Numerico) + sizeof(Un_Mensaje.Mensaje),2, 0);

          printf("%s",Un_Mensaje.Mensaje);

    }

    printf("\n");

    // Cerramos el fichero de texto
    fclose(archivo);

    // Se borra y cierra la cola de mensajes.
    // IPC_RMID indica que se quiere borrar. El puntero del final son datos
    // que se quieran pasar para otros comandos. IPC_RMID no necesita datos,
    // así que se pasa un puntero a NULL.
    msgctl (Id_Cola_Mensajes, IPC_RMID, (struct msqid_ds *)NULL);

    exit(0);
}
```

Donde la utilización y empleo de las colas de mensajes está explicado en los comentarios del código. Lo que realizamos es: Tomar línea a línea la información del fichero; Enviarla por la cola de mensajes; Recibirla por la cola de mensajes; Mostrarla por pantalla.

La aplicación con la que nos comunicaremos será la de cliente de eco, la cual necesita ciertas modificaciones. El código será el siguiente:

```c
//Cliente de eco sobre el protocolo TCP.

#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<stdlib.h>
#include<errno.h>
#include<sys/unistd.h>
#include<netdb.h>
#include<sys/ipc.h>
#include<sys/msg.h>

#define SERV_TCP_PORT 7500 // Puerto que utiliza el servidor para eco
#define SERV_HOST_ADDR "157.88.201.95"
#define MAXLINE 512
#define MAXTEXT 100

struct MiMensaje
{
 long Id_Mensaje;
 int Dato_Numerico;
 char Mensaje[MAXTEXT];
}Un_Mensaje;

main(int argc, char *argv[])
{
    //Variables de programa
    int sockfd, sockConect, sockWrite, sockRead, cerrarSocket;
    struct sockaddr_in serv_addr;

    //Ponemos a 0 la estructura de direcciones remotas
    bzero((char *) &serv_addr, sizeof(serv_addr));

    //Rellenamos la estructura de direcciones remotas
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port = htons(SERV_TCP_PORT);

    // Cola de mensajes
    key_t Clave1;
    int Id_Cola_Mensajes;

    // Obtengo una clave para la cola de mensajes
    Clave1 = ftok("/bin/ls", 35);
    // Compruebo errores
    if (Clave1 == (key_t)-1)
    {
        printf("Error al obtener clave para cola mensajes");
        return(0);
    }

    // Creo la cola de mensajes si no ha sido creada ya
    Id_Cola_Mensajes = msgget (Clave1, 0660 | IPC_CREAT);
    if (Id_Cola_Mensajes == -1)
    {
        printf("Error al obtener identificador para cola mensajes");
        return(0);
    }

    while(1)
    {

        // Recibimos el mensaje por la cola de mensajes
        // Leer un mensaje de cola 1
        msgrcv (Id_Cola_Mensajes, (struct msgbuf *)&Un_Mensaje,sizeof(Un_Mensaje.Dato_Numerico) + sizeof(Un_Mensaje.Mensaje), 1, 0);

        /*Creamos un socket para que el cliente pueda enviar y recibir mensajes
        AF_INET para sockets IPv4, SOCK_STREAM para sockets TCP y 0 porque es una aplicacion típica*/
        sockfd=socket(AF_INET,SOCK_STREAM,0);
        //Comprobamos errores
        if (sockfd<0)
        {
            perror("client: can't open stream socket");
            exit(EXIT_FAILURE);
        }
```

```
        //Conectamos el socket con el servidor
        sockConect=connect(sockfd,(struct sockaddr *) &serv_addr, sizeof(serv_addr));
        //Comprobamos errores
        if (sockConect < 0)
        {
            perror("client: can't connect to server");
            exit(EXIT_FAILURE);
        }

        //Escribimos en el socket
        sockWrite=write(sockfd, Un_Mensaje.Mensaje, sizeof(Un_Mensaje.Mensaje));
        //Comprobamos errores
        if (sockWrite < 0)
        {
            perror("writing on stream socket");
            exit(EXIT_FAILURE);
        }

        //Leemos el eco del socket
        sockRead=read(sockfd, Un_Mensaje.Mensaje, sizeof(Un_Mensaje.Mensaje));
        //Comprobamos errores
        if (sockRead < 0)
        {
            perror("reading stream message");
            exit(EXIT_FAILURE);
        }

         // Mensaje a enviar será de tipo 2
         Un_Mensaje.Id_Mensaje = 2;
         Un_Mensaje.Dato_Numerico = 13;

         // Enviamos por la cola de mensajes
         msgsnd (Id_Cola_Mensajes, (struct msgbuf *)&Un_Mensaje,sizeof(Un_Mensaje.Dato_Numerico)+sizeof(Un_Mensaje.Mensaje),IPC_NOWAIT);

         //Cerramos el socket
         cerrarSocket=close(sockfd);
         //Comprobamos errores
         if(cerrarSocket<0)
         {
             perror("close()");
             exit(EXIT_FAILURE);
         }
    }

    // Cerramos la cola de mensajes
    msgctl (Id_Cola_Mensajes, IPC_RMID, (struct msqid_ds *)NULL);

    exit(0);
}
```

Estas modificaciones se basan en añadir las instrucciones necesarias para la comunicación con el programa gestor mediante colas de mensajes. Más detalladamente hemos realizado las siguientes modificaciones (todo ello se explica e incluye en el código aportado):

1. Definición de la estructura llamada “struct MiMensaje”. Tendrá 3 campos con la intención de diferenciar tipos de mensajes e incluir los datos necesarios.
2. Definición y obtención de claves de la cola de mensajes.
3. Creación de la cola de mensajes para lectura y escritura.
4. Escribir las instrucciones necesarias para recibir y enviar información a través de la cola de mensajes.
5. Cerrar la cola de mensajes.

Como demostración del funcionamiento mostramos a continuación un ejemplo de la ejecución de ambas aplicaciones (gestor y cliente_b), obteniendo como resultado:

```
[cci48@carpanta P2]$ ./gestor

Nombre del fichero: Prueba.txt

CONTENIDO DEL FICHERO SELECCIONADO:

Primera linea
Segunda linea
Linea 3
Doble salto:

Fin doble salto
Fin
```

Donde podemos observar que funciona correctamente incluso los saltos de línea. Dentro de la carpeta comprimida que hemos subido para esta práctica, se incluye un archivo llamado “Prueba.txt” que tiene el contenido mostrado anteriormente. Para la correcta ejecución y funcionamiento se podrán ejecutar indistintamente primero cualquiera de ambas aplicaciones, ya que las colas de mensajes se gestionan internamente y se mantendrá en espera a la aplicación correspondiente.

Bibliografía

Nos hemos apoyado en las siguientes páginas de Internet:

1. Pubs.opengroup.org
2. www.ibm.com
3. www.wikibooks.es
4. Man7.org/linux
5. www.chuidiang.org
6. es.tldp.org
7. www.cs.rpi.edu
8. www.gta.ufrj.br
9. www.cs.rpi.edu
10. Stackoverflow.com
11. Linuxfocus.org