PROGRAMACIÓN DE SOCKETS EN ENTORNO UNIX

P.1.1

Laboratorio de Redes y Servicios

INTEGRANTES DEL GRUPO:

Nombre: MANUEL MONTOYA CATALÁ

Nombre: ANDRÉS BEATO OLLERO

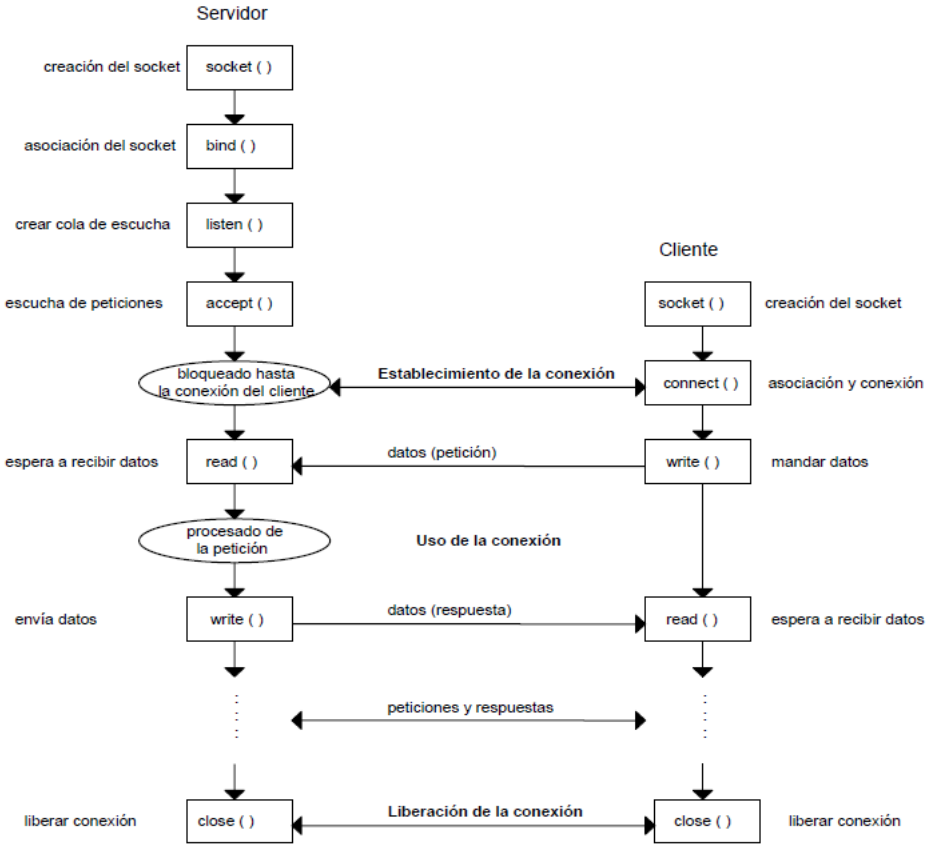
ÍNDICE

INTRODUCCIÓN

Un socket es una interfáz de comunicación entre dos procesos que se ejecutan en máquinas diferentes, provee un mecanismo para la transmisión de flujos de información entre máquinas conectadas a través de una red abstrayendo dicha red, reduciendo la comunicación al uso de un descriptor de fichero (puntero al archivo) donde al escribir sobre él, write(), estamos enviando dicha información a la otra máquina y al leer del mismo, read(), leemos la información que se nos ha enviado.
Dado que la mayoría de redes utilzan tecnología IP, nosotros nos dedicaremos a crear Sockets IP, para ello necesitaremos un API de Sockets, nosotros usaremos el API de Unix.
Un Socket IP necesita como parámetros mínimos:

- Dirección IP de la máquina objetivo (Ej. 198.168.1.23)
- Puerto de la máquina objetivo (Ej. 5003)

En función del protocolo de transporte utilizado, necesitaremos unos parámetros adicionales u otros, en esta práctica se utilizarán los protocolos TCP y UDP en modelo Cliente-Servidor, pero dado que los sockets TCP son los realmente utilizados en la práctica, a continuación se hace una breve explicación introductoria de un socket TCP.
El proceso de generación y uso de un socket de estas características se puede resumir en la siguiente imagen:



Con socket() creamos el mismo, indicando su tipo, en nuestro caso IP y TCP o UDP. Mediante bind() indicaremos al socket nuestra dirección (IP, puerto…) solo es necesario en caso de que queramos establecer dichos parámetros para que otro ordenador pueda contactarnos, no para enviar información.
Las funciones listen(), accept() y connect() solo se utilizan en comunicaciones orientadas a conexión y sirven para ponerse a la escucha en un puerto, aceptar una conexión entrante y establecer una conexión respectivamente. Una vez creado el socket, podemos enviar y recibir información como si de un fichero tipo Unix se tratase mediante write() y read(), sin embargo existen otras funciones como send() y recv() que tienen las mismas propiedades y pueden realizar ciertas funcionalidades extra por lo que son estas las que utilizaremos. Cabe resaltar que las funciones de lectura (read(), recv(), recvfrom()…) bloquean el flujo del programa a la espera de recibir datos por el socket a menos que les indiquemos lo contrario.

1. CREACIÓN DEL PING NO ORIENTADO A CONEXIÓN

En este apartado se ha realizado la programación de un servidor y cliente UDP que emulan el servicio PING basado en este protocolo, que aunque no es lo común se implementa en esta práctica por motivos didácticos. Para ello hemos hecho uso de la API de sockets de LINUX.
Hemos establecido un protocolo simple en el cual el cliente, al conectarse al servidor, envía un paquete de datos, el servidor se pondrá a la escucha y al recibirlo reenviará al cliente un mensaje con el mismo contenido que ha recibido para hacer ver en el cliente el correcto funcionamiento del ping. Tanto el cliente como el servidor se ejecutan mediante línea de comandos.

1.1 CONFIGURACIÓN PING UDP

a) Creación de socket

Se comienza llamando a la función de creación de sockets tanto en el cliente como en el servidor: socket(dominio, tipo, protocolo).
Esta función recibe como parámetros:

- dominio: El dominio del socket, en este caso pasamos como dominio AF_INET, dominio de Internet, donde se utilizan los protocolos TCP y UDP).
- tipo: Tipo de socket que vamos a crear. Utilizamos SOCK_DGRAM para UDP y SOCK_STREAM para TCP.
- protocolo: Protocolo utilizado en el socket. Con 0 utilizamos el protocolo por defecto.

b) Enlace del socket

Para asociar dirección IP y puerto al socket creado se debe llamar a la función bind(desc, p_direccion, long). Esta función recibe como parámetros:

- desc : El descriptor del socket creado.
- *p_direccion: Puntero a la estructura que contiene la dirección IP y el puerto a asignar.
- long: Longitud de la dirección.

En el cliente UDP no es necesario llamar a la función bind(), la aportación de la llamada a esta función en el cliente es que debido a que el protocolo UDP es no orientado a conexión si dos clientes hablan al servidor a la vez, si este no sabe el puerto desde el que le llega la petición no va saber dónde tiene que contestar.
Como en nuestro caso el ping debe realizará una sola conexión de prueba con nuestro servidor hemos optado por no hacer esta llamada en el cliente UDP.
En el servidor llamamos a esta función después de rellenar la estructura "sockaddr_in dir_serv" en la que se almacena la dirección IP del servidor y el puerto del servidor.

c) Envío y recepción de datos

El envió de datos se realiza con la función sendto(desc, msg, lg, opcion, p_dest, lg_dest).
Esta función recibe como parámetros:

- desc: El descriptor del socket creado.
- *msg: dirección del mensaje a enviar.
- lg: longitud del mensaje a enviar.
- opcion: 0
- *p_dest: puntero a la estructura con la dirección a la que enviar los datos.
- lg_dest: longitud de la dirección destino.

La recepción de datos se realiza con la función recvfrom(desc, msg, lg, opcion, p_exp, p_lgexp). Esta función recibe como parámetros:

- desc: El descriptor del socket creado.
- *msg: dirección del buffer en el que se guardan los datos recibidos.
- lg: longitud del mensaje recibido.
- opcion: 0
- *p_expt: puntero a la estructura con la dirección de la cual proceden los datos.
- p_ldexp: puntero a la variable que contiene la longitud de la estructura que almacena la dirección de los datos recibidos.

d) Cierre del socket

Con la llamada a la función close(desc), se cierra el socket que había creado. Se pasa como parámetro a esta función el descriptor del socket que se debe cerrar.

1.2 SERVIDOR

Para el uso del servidor tecleamos:

./ping_noc_serv Puerto

El servidor establecerá un socket UDP a la escucha en el puerto dado como parámetro, después entrará en un bucle infinito en el cual esperará conexiones entrantes y responderá reenviando el mensaje que ha llegado desde el cliente. Como características adicionales el programa comprueba que se hayan dado los parámetros necesarios y nos informa de la IP del cliente, del número de bytes recibidos y del contenido del mensaje recibido.

El código del servidor **ping_oc_serv.c** es:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/types.h>

#define MAXLINEA 1024

int main(int argc, char *argv[])
{
        int sockfd; // descriptor de socket
        struct sockaddr_in dir_serv; // estructura que almacena la direccion IP y numero de puerto
del servidor
        struct sockaddr_in dir_client; // estructura que almacena la direccion IP y numero de
puerto del cliente

        int addr_len, numbytes,numbytes_env;
        char buf[MAXLINEA]; // Datos recibidos
        if (argc != 2) { //comprobamos que no haya error en la entrada de parametros al main
        printf("Error en la entrada de parametros a la funcion\n");
        exit(1);
        }

        /* Llamada a la funcion de creacion del socket */
        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
              perror("socket");
              exit(1);
        }

        /* Rellenamos la estructura del servidor */
        dir_serv.sin_family = AF_INET;
```

```
        dir_serv.sin_port = htons(atoi(argv[1]));
        dir_serv.sin_addr.s_addr = INADDR_ANY; // escuchamos en todas las IPs
        bzero(&(dir_serv.sin_zero), 8); //ponemos a 0 el resto de parametros de la estructura

        /* Llamamos a la funcion bind() */
        printf("Socket creado\n");
        if (bind(sockfd, (struct sockaddr *)&dir_serv, sizeof(struct sockaddr)) == -1) {
              perror("bind");
              exit(1);
        }

        /* Se reciben los datos */
        addr_len = sizeof(struct sockaddr);
        printf("Esperando datos ...\n");
        if ((numbytes=recvfrom(sockfd, buf, MAXLINEA, 0, (struct sockaddr *)&dir_client, (socklen_t
*)&addr_len))          == -1)
        {
        perror("recvfrom");
        exit(1);
        }
        /*Reenviamos lo que hemos recibido para que en el cliente se conozca el buen funcionamineto
del ping*/
        if ((numbytes_env=sendto(sockfd,buf,MAXLINEA,0,(struct sockaddr *)&dir_client,
sizeof(struct          sockaddr))) == -1)
        {
               perror("sendto");
               exit(1);
        }

        /* Se visualiza lo recibido */
        printf("paquete proveniente de : %s\n",inet_ntoa(dir_client.sin_addr));
        printf("longitud del paquete en bytes : %d\n",numbytes);
        buf[numbytes] = '\0';
        printf("el paquete contiene : %s\n", buf);

        /* cerramos descriptor del socket */
        close(sockfd);

        return 0;
}
```

1.3 CLIENTE

Para el uso del servidor tecleamos:

./ping_noc Server_addr Server_port [mensaje a enviar]

El cliente establecerá un socket con el que se conectará al servidor dado por Server_addr en su puerto Server_port, y transmitirá el mensaje que se introduce como parámetro. El servidor puede ser dado tanto como IP como por su URL. Como características adicionales el programa comprueba que se hayan dado los parámetros necesarios y nos informa del número de bytes enviados, de la IP y puerto de la máquina del servidor. Para saber que el ping ha funcionado de forma correcta el servidor devolverá el mismo mensaje que hemos enviado desde el cliente y este será mostrado por pantalla.
Al acabar de enviar y recibir los paquetes, el cliente cierra el socket y termina.

El código del cliente **ping_noc_serv.c** es:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/types.h>

#define MAXLINEA 1024

int main(int argc, char *argv[])
{
        int sockfd; //descriptor de socket
        struct sockaddr_in dir_serv; // estructura que almacena la direccion IP y numero de puerto
del servidor
        struct hostent *he; //estructura para guardar la direccion IP a partir del nombre del host
        int numbytes,numbytes_rec,addr_len; //nº de bytes enviados al servidor
        char buf[MAXLINEA]; // Datos recibidos
        if (argc != 4) { //comprobamos que no haya error en la entrada de parametros al main
        printf("Error en la entrada de parametros a la funcion\n");
        exit(1);
        }

        /* Convertimos el hostname a su direccion IP */
        if ((he=gethostbyname(argv[1])) == NULL) {
              herror("gethostbyname");
              exit(1);
        }

        /* Llamada a la funcion de creacion del socket */
        if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
              perror("socket");
              exit(1);
        }

        /* Rellenamos la estructura del servidor */
        dir_serv.sin_family = AF_INET;
        dir_serv.sin_port = htons(atoi(argv[2]));
        dir_serv.sin_addr = *((struct in_addr *)he->h_addr);//introducimos la IP desde gethostname
        bzero(&(dir_serv.sin_zero), 8);//ponemos a 0 el resto de parametros de la estructura

        /* Envio de datos al servidor */
        if ((numbytes=sendto(sockfd,argv[3],strlen(argv[3]),0,(struct sockaddr *)&dir_serv,
sizeof(struct           sockaddr))) == -1)
        {
               perror("sendto");
               exit(1);
        }
        /*Recibimos los datos del servidor para conocer que el ping llega correctamente*/
        addr_len = sizeof(struct sockaddr);
        if ((numbytes_rec=recvfrom(sockfd, buf, MAXLINEA, 0, (struct sockaddr *)&dir_serv,
(socklen_t *)&addr_len))              == -1)
        {
        perror("recvfrom");
        exit(1);
        }
        printf("Enviados %d bytes hacia IP %s y puerto %s
\n",numbytes,inet_ntoa(dir_serv.sin_addr),argv[2]);

        buf[numbytes] = '\0';
        printf("Respuesta: eco %s \n", buf);
        /* Cerramos el socket */
        close(sockfd);

        return 0;
}
```

1.4 MAKEFILE

```
all: ping_noc ping_noc_serv

ping_noc.o: ping_noc.c
	gcc -c ping_noc.c -Wall

ping_noc: ping_noc.o
	gcc -o ping_noc ping_noc.o -Wall

ping_noc_serv.o: ping_noc_serv.c
	gcc -c ping_noc_serv.c -Wall

ping_noc_serv: ping_noc_serv.o
	gcc -o ping_noc_serv ping_noc_serv.o -Wall
```

1.5 EJEMPLO DE UTILIZACIÓN

Para comprobar el funcionamiento del ping UDP se ejecutan ambos archivos por separado en la línea de comandos. Establecemos el servidor escuchando en el puerto 6001 y desde el cliente enviamos un paquete a la dirección localhost (que será traducida a la dirección IP 127.0.0.1 internamente por el cliente), al puerto 6001) y cuyo contenido se pasa como parámetro en la línea de comandos.
Como podemos observar el cliente nos indica el número de bytes enviados, la dirección y el puerto de envío y el mensaje de respuesta del servidor, que por como esta implementada nuestra aplicación debe coincidir con el enviado por nosotros mismos.
Por su parte el servidor informa del número de bytes recibidos y de la dirección IP y puerto origen.

- Parte Cliente:

```
andres@andres-K53SC:~/Escritorio/udp_prueba$ ./ping_noc localhost 6001 prueba-udp
Enviados 10 bytes hacia IP 127.0.0.1 y puerto 6001
Respuesta: eco prueba-udp
andres@andres-K53SC:~/Escritorio/udp_prueba$
```

- Parte servidor:

```
andres@andres-K53SC:~/Escritorio/udp_prueba$ ./ping_noc_serv 6001
Socket creado
Esperando datos ...
paquete proveniente de : 127.0.0.1
longitud del paquete en bytes : 10
el paquete contiene : prueba-udp
andres@andres-K53SC:~/Escritorio/udp_prueba$
```

2. CREACIÓN DEL PING ORIENTADO A CONEXIÓN

En este apartado se ha realizado la programación de un servidor y cliente TCP que emulan el servicio PING. Para ello hemos hecho uso de la API de sockets de LINUX.
Hemos establecido un protocolo simple en el cual el cliente, al conectarse al servidor, envía un primer paquete donde indica al servidor el número de paquetes que van a ser enviados, sin indicar su tamaño, el servidor se pondrá a la escucha de dichos paquetes y al recibir cada uno de ellos enviará un paquete con el número del paquete recibido. El cliente no envía un paquete hasta que no reciba la respuesta del anterior y para cada uno de ellos calcula el retardo de ida y vuelta del paquete. Tanto el cliente como el servidor se ejecutan mediante línea de comandos.

2.1 SERVIDOR

Para el uso del servidor tecleamos:

./ping_oc_serv Puerto

El servidor establecerá un socket TCP a la escucha en el puerto dado como parámetro, después entrará en un bucle infinito en el cual esperará conexiones entrantes y responderá según nuestro protocolo. Como características adicionales el programa comprueba que se hayan dado los parámetros necesarios y nos informa de la IP y puerto de la máquina cliente.

El código del servidor **ping_oc_serv.c** es:

```
#include "sockets_cab.h"

int main(int argc, char **argv) {

        char * Server_port;
        int i;
        struct sockaddr_in Server_addr, Clien_addr;
        int     descSocket, nuevoDescSocket;
        int     longDirCliente ;       // longitud de la direccion del cliente
        int     num_pkt = 0 ;
        char aux_buff;                 // Sitio donde leer por lo menos un byte de los recibidos

        // ********* Process the given parameters  ********
        if (argc < 2) {
                printf("Not enough parameters \n");
                exit(-1);
        }
        Server_port = argv[1];

        // Open the socket and checks it went well
        descSocket = socket(AF_INET, SOCK_STREAM, 0); // Socket IP, TCP
        if (descSocket < 0) {
                fprintf(stderr, "SERVIDOR: no se ha podido abrir el socket \n") ;
                exit(-1) ;
        }
        // Initialize the server addr structure
        bzero ((char*) &Server_addr, sizeof(Server_addr));
        Server_addr.sin_family = AF_INET;
        Server_addr.sin_addr.s_addr= htonl(INADDR_ANY);
        Server_addr.sin_port = htons(atoi(Server_port));

        // Stablish TCP port of the server
        if (bind(descSocket,(struct sockaddr *) &Server_addr,sizeof(Server_addr))< 0) {
                fprintf(stderr, "SERVIDOR: error al vincular la direccion local\n") ;
                exit(-1) ;
        }

        // Listen to incoming TCP connections (5 max in the queue)
        listen(descSocket, 5);
```

```
        while (1) {
                printf("Servidor escuchando en el puerto %s \n", Server_port) ;
                nuevoDescSocket= accept(descSocket,(struct sockaddr *) &Clien_addr ,&longDirCliente);
                if (nuevoDescSocket < 0) {
                        fprintf(stderr, "SERVIDOR: error al aceptar nueva conexion \n") ;
                        exit(-1) ;
                }
                printf("Conexion establecida con %s en el puerto %i \n",
                        inet_ntoa(((struct in_addr)Clien_addr.sin_addr)),Clien_addr.sin_port);

                // Leemos el numero de paquetes recibidos
                recv(nuevoDescSocket, &num_pkt, sizeof(int),0);
                printf("***** Recibiremos %d paquetes *****\n", num_pkt);

                for (i = 0; i < num_pkt; i++){
                        recv(nuevoDescSocket, &aux_buff, sizeof(char),0);
                        send(nuevoDescSocket, &i, sizeof(int),0);
                        fprintf(stderr, "SERVIDOR: Ping %i Respondido \n", i +1) ;
                }
                        printf("***** Transmision terminada *****\n");
        }
                close(descSocket);
                close(nuevoDescSocket);
}
```

2.2 CLIENTE

Para el uso del servidor tecleamos:

./ping_oc Server_addr Server_port [num_pkt] [bytes_pkt]

El cliente establecerá un socket con el que se conectará al servidor dado por Server_addr en su puerto Server_port, y comenzará a transmitir según el protocolo anteriormente descrito 5 paquetes de 64 bytes con datos de relleno '**$**'. El servidor puede ser dado tanto como IP como por su URL. Como características adicionales el programa comprueba que se hayan dado los parámetros necesarios y nos informa de la IP y puerto de la máquina del servidor. Además los últimos 2 parámetros son adicionales y sirven para enviar otro número y tamaño de paquetes deseado.

Antes de enviar un paquete y después de recibir su respuesta, el programa usará la llamada al sistema gettimeofday() para obtener el tiempo del sistema en microsegundos, la resta de los mismos nos dará el retardo de ida y vuelta. Al acabar de enviar y recibir los paquetes, el cliente cierra el socket y termina.
El código del cliente **ping_oc_serv.c** es:

```
#include "sockets_cab.h"

int main(int argc, char **argv) {
        int errores,i;
        char * Server_id; // IP or name of the server we will connect to
        char * Server_port; // Port of the server we will connect to;

        struct  sockaddr_in Server_addr ;     // Structure with the data about socket
        int     descSocket ;                            // Socket descriptor
        // Time struct (time.h) to get the the time in microseconds
        struct timeval time_sent, time_recv;
        int delay_ping;                                 // Return time delay
        int num_pkt = 5;                        // Number of packets we will send
        int bytes_per_pkt = 64;                 // Bytes per packet

        char *send_buffer;                      // Send buffer
        int Server_response;                // Server Response

        struct hostent *he;                 // Holds the addr of Server in case we are given an URL

        // ********* Process the given parameters  ********
        if (argc < 3) {                 // If we are not given the basic parameters
                printf("Not enough parameters \n");
                exit(-1);
        }

        Server_id = argv[1];            // Get the server addr
```

```
        if ((Server_id[0] < '0')||(Server_id[0] > '9')){ // If we are given a name and not IP
                if ((he =gethostbyname(argv[1])) == NULL) {
                        herror("gethostbyname");
                        exit(1);
                }
                Server_id = inet_ntoa(*((struct in_addr *)he->h_addr)) ;
        }
        Server_port = argv[2];

        if (argc > 3){          // If we are given optional parameters
                num_pkt = atoi(argv[3]);
        }

        if (argc > 4){          // If we are given optional parameters
                bytes_per_pkt = atoi(argv[4]);
        }

        // Open the socket and checks it went well
        descSocket = socket(AF_INET, SOCK_STREAM, 0); // Socket IP, TCP
        if (descSocket== -1) {
                fprintf(stderr, "CLIENTE: No se ha podido abrir el socket \n") ;
                exit(-1) ;
        }
        fprintf(stderr, "CLIENTE: Socket abierto \n") ;

        // Initialize the Server addr structure to connect server
        bzero((char *) &Server_addr, sizeof(Server_addr));  // Set everything to 0
        Server_addr.sin_family = PF_INET;                   //
        Server_addr.sin_addr.s_addr= inet_addr(Server_id);  // IP of the machine we will connect with
        Server_addr.sin_port = htons(atoi(Server_port));    // Port of the machine we will connect to

        // Connect to the server
        errores = connect(descSocket,(struct sockaddr *) &Server_addr, sizeof(Server_addr));
        if ( errores ==-1) {
                fprintf(stderr, "CLIENTE: No se ha podido conectar con servidor\n") ;
                exit(-1) ;
        }
        printf("CLIENTE: Conexion establecida con %s en el puerto %s \n", Server_id, Server_port) ;

        // Reserve dynamic memory for the buffer and initialice it to '$'
        send_buffer = (char *) malloc (bytes_per_pkt*sizeof(char));
        for (i = 0 ; i < bytes_per_pkt; i++){
                send_buffer [i] = '$';
        }

        // Send the data throught the socket
        send(descSocket, &num_pkt, sizeof(int),0) ;
        printf("CLIENTE: Enviando %d paquetes de %d bytes al servidor \n", num_pkt,bytes_per_pkt) ;
        for (i = 0;i < num_pkt; i++){
                // Get the time when we send
                errores = gettimeofday( &time_sent, NULL);
                if (errores==-1) {
                        perror("CLIENTE: Error al llamar a la fecha:");
                        exit(1);
                }
                // Send and receive data to server
                send(descSocket, send_buffer, bytes_per_pkt*sizeof(char),0) ;
                recv (descSocket, &Server_response, sizeof(Server_response),0) ;
                printf("Respondido el envio %i    ",Server_response+1) ;
                // Get the time when we receive
                errores = gettimeofday( &time_recv, NULL);
                if (errores==-1) {
                        perror("Error al llamar a la fecha:");
                        exit(1);
                }
                delay_ping  = (int)(time_recv.tv_usec - time_sent.tv_usec);
                printf("Delay del ping %i \n",delay_ping) ;
        }
        close(descSocket); // Close the socket connection
        free(send_buffer); // Free memory
}
```

2.3 MAKEFILE Y ARCHIVO DE CABECERAS

Los siguientes archivos también son utilizados para la generación del proyecto:
Código del archivo de cabeceras **sockets_cab.h**:

```
#ifndef __DEFS_H__
#define __DEFS_H__
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <time.h>
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>

#endif
```

Código del **Makefile**:

```
all: ping_oc ping_oc_serv

ping_oc_serv.o: ping_oc_serv.c
	gcc -c ping_oc_serv.c -Wall
ping_oc_serv: ping_oc_serv.o
	gcc -o ping_oc_serv ping_oc_serv.o -Wall

ping_oc.o: ping_oc.c
	gcc -c ping_oc.c -Wall
ping_oc: ping_oc.o
	gcc -o ping_oc ping_oc.o -Wall
```

2.4 EJEMPLO DE UTILIZACIÓN

Para demostrar el funcionamiento del código hemos realizado una conexión entre cliente y servidor en 2 máquinas diferentes conectadas a través de la red eduroam de la UAH. El servidor, la máquina 172.22.29.90 se pondrá a escuchar en el puerto 5001 y el cliente, la máquina 172.22.30.214 se conectará a él y le enviará 3 paquetes de 64 Kbytes

- Parte servidor:

```
Servidor escuchando en el puerto 5001
Conexion establecida con 172.22.30.214 en el puerto 62656
***** Recibiremos 3 paquetes *****
SERVIDOR: Ping 1 Respondido
SERVIDOR: Ping 2 Respondido
SERVIDOR: Ping 3 Respondido
***** Transmision terminada *****
Servidor escuchando en el puerto 5001
```

- Parte cliente:

```
manuel@ubuntu:~/Desktop/TCP/ping$ ./ping_oc 172.22.29.90 5001 3 64
CLIENTE: Socket abierto
CLIENTE: Conexion establecida con 172.22.29.90 en el puerto 5001
CLIENTE: Enviando 3 paquetes de 64 bytes al servidor
Respondido el envio 1   Delay del ping 409735
Respondido el envio 2   Delay del ping 93042
Respondido el envio 3   Delay del ping 169
```

2.5 FUNCIONAMIENTO CON WIRESHARK

Para demostrar el correcto funcionamiento de los sockets IP hemos capturado los paquetes de la comunicación anterior mediante la herramienta Wireshark. La comunicación está formada por los siguientes paquetes:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.190.129	172.22.29.90	TCP	74	51192 > 5001 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1 TSval=280517 TSecr=0 WS=16
2	0.006417	172.22.29.90	192.168.190.129	TCP	60	5001 > 51192 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
3	0.006454	192.168.190.129	172.22.29.90	TCP	54	51192 > 5001 [ACK] Seq=1 Ack=1 Win=14600 Len=0
4	0.006563	192.168.190.129	172.22.29.90	TCP	58	51192 > 5001 [PSH, ACK] Seq=1 Ack=1 Win=14600 Len=4
5	0.006710	172.22.29.90	192.168.190.129	TCP	60	5001 > 51192 [ACK] Seq=1 Ack=5 Win=64240 Len=0
6	0.006769	192.168.190.129	172.22.29.90	TCP	118	51192 > 5001 [PSH, ACK] Seq=5 Ack=1 Win=14600 Len=64
7	0.006858	172.22.29.90	192.168.190.129	TCP	60	5001 > 51192 [ACK] Seq=1 Ack=69 Win=64240 Len=0
8	0.416150	172.22.29.90	192.168.190.129	TCP	60	5001 > 51192 [PSH, ACK] Seq=1 Ack=69 Win=64240 Len=4
9	0.416227	192.168.190.129	172.22.29.90	TCP	54	51192 > 5001 [ACK] Seq=69 Ack=5 Win=14600 Len=0
10	0.416421	192.168.190.129	172.22.29.90	TCP	118	51192 > 5001 [PSH, ACK] Seq=69 Ack=5 Win=14600 Len=64
11	0.416791	172.22.29.90	192.168.190.129	TCP	60	5001 > 51192 [ACK] Seq=5 Ack=133 Win=64240 Len=0
12	0.509360	172.22.29.90	192.168.190.129	TCP	62	5001 > 51192 [PSH, ACK] Seq=5 Ack=133 Win=64240 Len=8
13	0.509494	192.168.190.129	172.22.29.90	TCP	118	51192 > 5001 [PSH, ACK] Seq=133 Ack=13 Win=14600 Len=64
14	0.509679	192.168.190.129	172.22.29.90	TCP	54	51192 > 5001 [FIN, ACK] Seq=197 Ack=13 Win=14600 Len=0
15	0.511128	172.22.29.90	192.168.190.129	TCP	60	5001 > 51192 [ACK] Seq=13 Ack=197 Win=64240 Len=0

- Los 3 primeros paquetes son el three-way handshake entre lás maquinas para establecer la conexión TCP, está formado por el SYN, SYN ACK y ACK.
- Los siguiente 2 paquetes son el envió del cliente al servidor del número de paquetes que serán enviados según nuestro protocolo y la respuesta TCP del mismo.

```
▸ Frame 4: 58 bytes on wire (464 bits), 58 bytes captured (464 bits)
▸ Ethernet II, Src: Vmware_5a:c5:1d (00:0c:29:5a:c5:1d), Dst: Vmware_ee:7c:75 (00:50:56:ee:7c:75)
▸ Internet Protocol Version 4, Src: 192.168.190.129 (192.168.190.129), Dst: 172.22.29.90 (172.22.29.90)
▸ Transmission Control Protocol, Src Port: 51192 (51192), Dst Port: 5001 (5001), Seq: 1, Ack: 1, Len: 4
▾ Data (4 bytes)
    Data: 03000000
    [Length: 4]
```

Como vemos hemos indicado que vamos a enviar 3 paquetes. Podemos observar también que los datos se envían en modo Little Endian.

- Los paquetes 6 al 13 son el envío de los 3 paquetes de 64 bytes del servidor, las respuestas del servidor y los ACK del protocolo TCP a dicho paquetes. Como ejemplo de paquete Cliente-Servidor tenemos el frame 6 donde podemos observar los 64 caracteres '$' que envía nuestro protocolo.

```
▸ Frame 6: 118 bytes on wire (944 bits), 118 bytes captured (944 bits)
▸ Ethernet II, Src: Vmware_5a:c5:1d (00:0c:29:5a:c5:1d), Dst: Vmware_ee:7c:75 (00:50:56:ee:7c:75)
▸ Internet Protocol Version 4, Src: 192.168.190.129 (192.168.190.129), Dst: 172.22.29.90 (172.22.29.90)
▸ Transmission Control Protocol, Src Port: 51192 (51192), Dst Port: 5001 (5001), Seq: 5, Ack: 1, Len: 64
▾ Data (64 bytes)
    Data: 2424242424242424242424242424242424242424242424...
    [Length: 64]

0030  39 08 48 f5 00 00 24 24  24 24 24 24 24 24 24 24   9.H...$$ $$$$$$$$
0040  24 24 24 24 24 24 24 24  24 24 24 24 24 24 24 24   $$$$$$$$ $$$$$$$$
0050  24 24 24 24 24 24 24 24  24 24 24 24 24 24 24 24   $$$$$$$$ $$$$$$$$
0060  24 24 24 24 24 24 24 24  24 24 24 24 24 24 24 24   $$$$$$$$ $$$$$$$$
```

Como paquetes de respuesta del servidor tenemos el frame 12 donde como podemos observar el socket TCP ha unido lo que en principio eran dos mensajes de respuesta separados en un solo mensaje, respondiendo a los paquetes 1 y 2:

```
▸ Frame 12: 62 bytes on wire (496 bits), 62 bytes captured (496 bits)
▸ Ethernet II, Src: Vmware_ee:7c:75 (00:50:56:ee:7c:75), Dst: Vmware_5a:c5:1d (00:0c:29:5a:c5:1d)
▸ Internet Protocol Version 4, Src: 172.22.29.90 (172.22.29.90), Dst: 192.168.190.129 (192.168.190.129)
▸ Transmission Control Protocol, Src Port: 5001 (5001), Dst Port: 51192 (51192), Seq: 5, Ack: 133, Len: 8
▾ Data (8 bytes)
    Data: 0100000002000000
    [Length: 8]
```

- Los últimos paquetes pertenecen al cierre de la conexión TCP, cierre iniciado por el cliente.

BIBLIOGRAFIA

[1]. José M. Arco, Bernardo Alarcos, “Programación de aplicaciones en redes de comunicaciones bajo entorno Unix”, Servicio Publicaciones UAH, 1997.

[2]. Beej’s Guide to Network Programming Using Internet Sockets

[3]. Programación con Sockets. Proyecto UCUAUMA7. Universidad de Málaga