Programación Distribuida y Tiempo Real

Ulises Jeremias Cornejo Fandos[1]

[1] *Licenciatura en Informática, Facultad de Informática, UNLP*

compiled: September 12, 2019

1. Identifique similitudes y diferencias entre los sockets en C y en Java

Para la creación y utilización de sockets en C se utiliza la librería estandar *sys/socket.h*, mientras que en Java se utiliza la librería *java.net* para la creación de los mismos y algunos módulos provenientes de la librería *java.io* para interactuar con los mismos a modo de lectura y escritura.

Una diferencia muy marcada es el enfoque de implementación de los Sockets en ambos lenguajes. Por un lado tenemos el enfoque de Java, que cuenta con una clase *java.net.Socket*. La misma permite establecer una comunicación bidireccional entre dos procesos en una red dada (el programa que define el socket y otro en la red). Además, se cuenta con la clase *java.net.ServerSocket* para definir más directamente sockets que estén escuchando y aceptando conexiones desde procesos clientes.

Por otro lado, en C no se encuentra tal distinción. El modelo Cliente/Servidor puede ser implementado pero la librería no cuenta con funciones pensadas explicitamente para la realización de esta tarea. La librería *sys/socket.h* expone la función ***socket*** la cual permite crear un socket.

En C, todos los sockets son representados por file descriptors, de los cuales solo se conoce un número de identificación. Estos son utilizados de la misma manera que archivos de un file system, leyendo y escribiendo en ellos con las mismas operaciones. Adicionalmente se utilizan las operaciones *bind*, *listen* y *accept* sobre el socket servidor para darle una dirección, permitirle escuchar por conexiones y aceptarlas respectivamente.

2. Tanto en C como en Java (directorios csock-javasock):

2.A. ¿Por qué puede decirse que los ejemplos no son representativos del modelo c/s?

En el modelo cliente/servidor (c/s), el servidor está la espera continua de algún pedido de conexión por parte de algún cliente. Luego, el esquema general de trabajo en un modelo c/s dado el pedido de conexión es el siguiente: inicialización (o handshape), transferencia de datos, y finalización.

En este caso, el servidor cierra la conexión dada la primer request y envio de respuesta. No se respetan los pasos de conexión anteriormente mencionados, pues en el ejemplo solo se realiza la transferencia de datos, sin envio de handshape o envío de mensajes de cierre de conexión.

2.B. Muestre que no necesariamente siempre se leen/escriben todos los datos involucrados en las comunicaciones con una llamada read/write con sockets. Sugerencia: puede modificar los programas (C o Java o ambos) para que la cantidad de datos que se comunican sea de 10^3 , 10^4 , 10^5 y 10^6 bytes y contengan bytes asignados directamente en el programa (pueden no leer de teclado ni mostrar en pantalla cada uno de los datos del buffer), explicando el resultado en cada caso. Importante: notar el uso de "attempts" en "...attempts to read up to count bytes from file descriptor fd..." así como el valor de retorno de la función read (del man read).

Se modificó la implementación del cliente y servidor de forma tal que el tamaño del buffer pueda ser definido en tiempo de compilación. (*Ver implementación en la sección 6.A*).

El tamaño del buffer tanto del cliente como del servidor se fue modificando con diferentes valores para ver hasta qué cantidad de datos podía leer, se probó con los siguientes valores: 10^3 , 10^4 , 10^5 y 10^6.

El servidor pudo leer correctamente todos los caracteres para un buffer size menor o igual a 10^4. Sin embargo, cuando se probó con un buffer size de 10^5, solo pudo leer ≈ 65480 caracteres.

2.C. Agregue a la modificación anterior una verificación de llegada correcta de los datos que se envían (cantidad y contenido del buffer), de forma tal que se asegure que todos los datos enviados llegan correctamente, independientemente de la cantidad de datos involucrados.

Para realiza esta implementación see agregó en el cliente y servidor una función de hashing, `unsigned long djb2(unsigned char *str);`, la cual genera un hash de los datos recibidos para después utilizarlo al realizar un checksum. (*Ver implementación en*

la sección 6.B).

El cliente genera un hash de los datos que se van a enviar. Una vez generado, se envían los datos, y se espera la respuesta del servidor para saber si los mismos fueron recibidos. Luego se envía el hash de los datos y el tamaño de los datos enviados. (*Ver 6.B.1*).

Por otra parte, el servidor espera los datos, el hash y el tamaño de los datos enviados por el cliente. Una vez que tiene todo esto, genera el hash de los datos recibidos y compara el hash y la cantidad de datos recibidos con el hash que se recibió y el tamaño de los datos que envia el cliente. (*Ver 6.B.2*).

2.D. Grafique el promedio y la desviación estándar de los tiempos de comunicaciones de cada comunicación. Explique el experimento realizado para calcular el tiempo de comunicaciones.

Para medir los tiempos de ejecución se utiliza la función *dwalltime* para medir el tiempo existente entre el comienzo de la transmisión de datos y la confirmación del servidor de los mismos fueron recibidos. (*Ver implementación en la sección 6.C*).

Para la realización de los experimentos se miden los tiempos de transmisión de 10^3, 10^4, 10^5 y 10^6 bytes. Cada una de estas configuraciones se ejecutan 10 veces.

Luego, los tiempos recolectados en las pruebas se utilizan para calcular la media y desviación estándar para el envío de cada cantidad de datos.

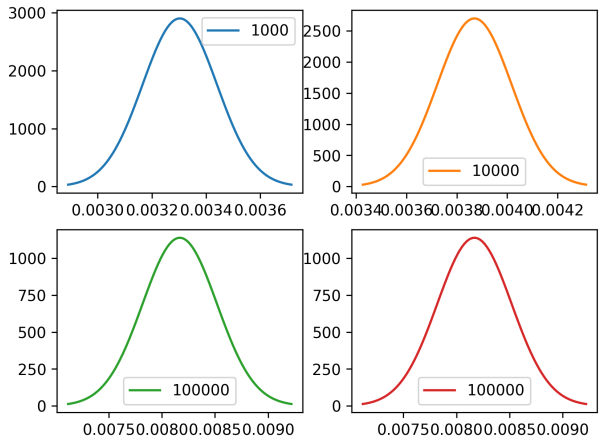

Fig. 1. Gráficas de las curvas gaussianas para cada escenario utilizando buffer size de 10^3, 10^4, 10^5 y 10^6.

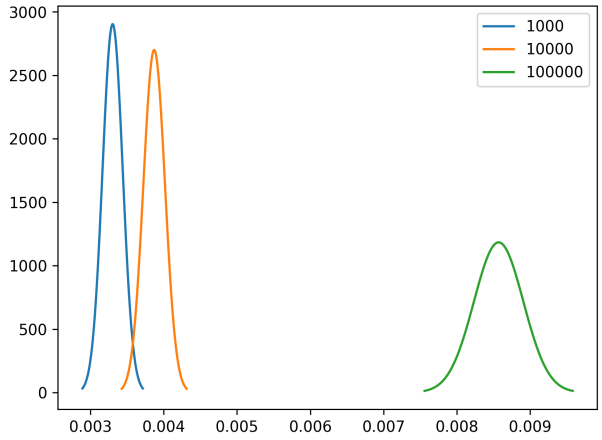

Fig. 2. Gráfica comparativa de las curvas gaussianas para cada escenario utilizando buffer size de 10^3, 10^4 y 10^5.

3. ¿Por qué en C se puede usar la misma variable tanto para leer de teclado como para enviar por un socket? ¿Esto sería relevante para las aplicaciones c/s?

El header o prototipo de la función *write* en C está dado por la siguiente expresión:

```
size_t write(int fd, void *buf, size_t nbytes);
```

donde *buf* es un puntero al espacio de memoria de *nbytes* bytes a partir del cual la system call obtendrá la información para escribir en el file descriptor.

Luego, el valor enviado como segundo argumento de la función *write* será cualquier puntero a un espacio de memoria, sea dinámica o estática. Es decir, podría enviar una referencia a un espacio de enteros (`int *`), espacio de caracteres (`char *`), etc.

En el caso de la variable utilizada para leer de teclado, la misma se define como sigue: `char buffer[256];`

Por lo tanto, buffer es una variable de tipo puntero, la cual referencia un espacio de memoria de caracteres alocado en forma estática.

Es por esto que, dado que *write* espera un argumento de tipo puntero y la variable *buffer* tiene ese tipo, puedo utilizar la misma variable tanto para entrada de texto del teclado como para el envio del mismo por un socket.

Esta potencia al momento de manejar memoria y punteros tiene ciertas desventajas al momento de programar una arquitectura como el modelo c/s. Se deben contar con precauciones dado que, siendo siempre un espacio de bytes, dependerá de como el cliente interprete los datos presentes en el espacio de memoria recibido al momento de acceder a la información.

4. ¿Podría implementar un servidor de archivos remotos utilizando sockets? Describa brevemente la interfaz y los detalles que considere más importantes del diseño. No es necesario implementar.

Para plantear una implementación de servidor de archivos remotos utilizando sockets puedo basar mi solución en dos protocolos existente que definen una interfaz para este tipo de implementaciones, **FTP** y **TFTP**.

- **FTP**

  Una implementación de FTP puede realizarse utilizando dos sockets de tipo TCP para el servidor. Los mismos pueden escuchar en los puertos estandar, 20 y 21, para datos y control respectivamente.

  Se cuenta con funciones específicas, entre otras las cuales se ejecuten sobre el *filesystem* del servidor remoto FTP.

- **TFTP**

  Por otro lado, una implementación de TFTP se implementa como un servidor que utiliza unicamente un socket de tipo UDP. El conjunto de instrucciones soportadas por el mismo serán menores a las soportadas por un servidor FTP.

Para la implementación propia de un servidor de archivos remotos opto por una alternativa similar a la que se plantea sobre un servidor FTP. Para realizar un servidor de este estilo se debe implementar una conexión TCP, es decir, un inicio y cierre de conexión de 3 vías o *3-ways-handshape*.

Las instrucciones soportadas por el servidor implementado podrían ser las siguiente:

- **ls**

  Crea una lista de todos los archivos que se encuentran en el directorio actual.

  El cliente envía el comando *ls* y luego, el servidor contesta con la lista de archivos que posee. Se debe verificar el tamaño de los datos enviados desde el servidor para asegurarse que lleguen todos.

- **pwd**

  Muestra el path absoluto del directorio actual.

  El cliente envia el comando *pwd* y el servidor responde con el path absoluto del duirectorio actual de trabajo.

- **cd**

  El comando significa change directory (cambiar el directorio) y se usa para pasar a un directorio diferente. El comando **cd** sin argumentos se utiliza para tener acceso al directorio principal.

  El cliente envia el comando *cd* con o sin argumentos y el servidor actualiza internamente el directorio de trabajo.

- **mkdir**

  El comando mkdir se utiliza para crear un directorio dentro del directorio actual. El uso de este comando se reserva para los usuarios que tengan acceso permitido.

- **get**

  Este comando permite recuperar un archivo que se encuentra en el servidor.

  El cliente envia el comando *get*. Si el comando aparece seguido del nombre de un archivo, el archivo remoto se transfiere a la máquina local, dentro del directorio local actual. Si aparece seguido de dos nombres de archivos, el archivo remoto (el primer nombre) se transfiere a la máquina local en el directorio local actual con el nombre del archivo especificado (el segundo nombre). Si el nombre del archivo contiene espacios, será necesario introducido entre comillas.

  El servidor comprueba la existencia del archivo. Dada su existencia, se abre el archivo y comprueba su tamaño. Luego, genera un *checksum* del archivo. Por último, envía el archivo y el *checksum* por el socket para comprar si se recibió correctamente.

- **put**

  Este comando se utiliza para enviar un archivo local al servidor.

  Si el comando aparece seguido del nombre de un archivo, el archivo local se transfiere al servidor en el directorio remoto actual. Si el comando aparece seguido de dos nombres de archivos, el archivo local (el primer nombre) se transfiere al servidor en el directorio remoto actual, con el nombre del archivo especificado (el segundo nombre). Si el nombre del archivo contiene espacios, será necesario introducido entre comillas.

  Finalmente, con el objetivo de verificar que el archivo sea enviado correctamente, se comprueba el tamaño del archivo y se lo envía al servidor. Para realizar la comprobación, se envía un *checksum*, el cual el servidor debe contrastar contra un *checksum* del archivo que le llegó.

- **delete**

  Borra el archivo especificado en el servidor remoto.

  El cliente envía el comando *delete* seguido del nombre del archivo que desea eliminar. El servidor comprueba la existencia del archivo, y lo elimina.

5. Defina qué es un servidor con estado (stateful server) y qué es un servidor sin estado (stateless server).

- **Stateful Server**

  Un servidor con estado es aquel que mantiene el estado de la información del usuario en forma de sesiones. Este tipo de servidores recuerda los datos del cliente (estado) de una solicitud a la siguiente. Servidores con estado, almacenar estado de sesión. Por lo tanto, pueden realizar un seguimiento de qué clientes han abierto qué archivos, punteros de lectura y escritura actuales para archivos, qué archivos han sido bloqueados por qué clientes, etc.

- **Stateless Server**

  A diferencia de un servidor con estado, el servidor sin estado es aquel que no mantiene ningún estado de la información para el usuario. En este tipo de servidores, cada consulta es completamente independiente a la anterior.
  Sin embargo, los servidores sin estado pueden identificar al usuario si la solicitud al servicio incluye una identificación de usuario única que se asignó anteriormente al mismo. Ese identificador (ID) del usuario deberá pasarse en cada consulta, a diferencia del caso de los servidores con estado que mantienen este ID de usuario en la sesión y los datos de la solicitud no necesariamente deben contener este ID.

6. Apéndice

6.A. Ejercicio 2b

Modificación de los programas en C que permite variar el tamaño de los buffers en tiempo de compilación.

6.A.1. Cliente

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

/*
 * En tiempo de compilación se puede definir esta macro con un valor numérico >= 0
 *
 */
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1000
#endif

/*
 * Error
 * -- Muestra un mensaje de error y finaliza la ejecución del programa.
 *
 * @param char * msg, Mensaje a mostrar
 *
 * @return void
 *
 */
void
error(char *msg)
{
        perror(msg);
        exit(0);
}

int
main(int argc, char *argv[])
{
        int sockfd, portno, n;
        struct sockaddr_in serv_addr;
        struct hostent *server;

        char buffer[BUFFER_SIZE];

        if (argc < 3)
        {
                fprintf(stderr, "usage %s hostname port\n", argv[0]);
                exit(0);
        }

        /* TOMA EL NUMERO DE PUERTO DE LOS ARGUMENTOS */
        portno = atoi(argv[2]);

        /* CREA EL FILE DESCRIPTOR DEL SOCKET PARA LA CONEXION */
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

```
        /* AF_INET - FAMILIA DEL PROTOCOLO - IPV4 PROTOCOLS INTERNET */
        /* SOCK_STREAM - TIPO DE SOCKET */

        if (sockfd < 0)
                error("ERROR opening socket");

        /* TOMA LA DIRECCION DEL SERVER DE LOS ARGUMENTOS */
        server = gethostbyname(argv[1]);
        if (server == NULL)
        {
                fprintf(stderr, "ERROR, no such host\n");
                exit(0);
        }

        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;

        /* COPIA LA DIRECCION IP Y EL PUERTO DEL SERVIDOR A LA ESTRUCTURA DEL SOCKET */
        bcopy((char *) server->h_addr,
              (char *) &serv_addr.sin_addr.s_addr,
              server->h_length);
        serv_addr.sin_port = htons(portno);

        /* DESCRIPTOR - DIRECCION - TAMAÑO DIRECCION */
        if (connect(sockfd, (const struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
                error("ERROR connecting");

        *buffer = 'y';
        memset((buffer + 1), 'e', BUFFER_SIZE - 2);

        /* ENVIA UN MENSAJE AL SOCKET */
        n = write(sockfd, buffer, strlen(buffer));
        if (n < 0)
                error("ERROR writing to socket");
        bzero(buffer, BUFFER_SIZE);

        /* ESPERA RECIBIR UNA RESPUESTA */
        n = read(sockfd, buffer, BUFFER_SIZE - 1);
        if (n < 0)
                error("ERROR reading from socket");

        printf("%s\n", buffer);

        return 0;
}
```

6.A.2. Servidor

```
/* A simple server in the internet domain using TCP
   The port number is passed as an argument */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <getopt.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```c
void set_verbose_mode(int argc, char *argv[], int *verbose_flag_ptr);

/*
 * En tiempo de compilación se puede definir esta macro con un valor numérico >= 0
 *
 */
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1000
#endif


/*
 * Error
 * -- Muestra un mensaje de error y finaliza la ejecución del programa.
 *
 * @param char * msg, Mensaje a mostrar
 *
 * @return void
 *
 */
void
error(char *msg)
{
        perror(msg);
        exit(0);
}


int
main(int argc, char *argv[])
{
        static int verbose_flag;

        int sockfd, newsockfd, portno, clilen;
        char buffer[BUFFER_SIZE];
        struct sockaddr_in serv_addr, cli_addr;
        int n;

        if (argc < 2)
        {
                fprintf(stderr, "ERROR, no port provided\n");
                exit(1);
        }

        /* CREA EL FILE DESCRIPTOR DEL SOCKET PARA LA CONEXION */
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        /* AF_INET - FAMILIA DEL PROTOCOLO - IPV4 PROTOCOLS INTERNET */
        /* SOCK_STREAM - TIPO DE SOCKET */

        if (sockfd < 0)
                error("ERROR opening socket");

        bzero((char *) &serv_addr, sizeof(serv_addr));
        /* ASIGNA EL PUERTO PASADO POR ARGUMENTO */
        /* ASIGNA LA IP EN DONDE ESCUCHA (SU PROPIA IP) */
        portno = atoi(argv[1]);
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = INADDR_ANY;
```

```
        serv_addr.sin_port = htons(portno);

        set_verbose_mode(argc, argv, &verbose_flag);

        /* Instead of reporting '--verbose'
        and '--brief' as they are encountered,
        we report the final status resulting from them. */
        if (verbose_flag)
                puts("verbose flag is set");

        /* VINCULA EL FILE DESCRIPTOR CON LA DIRECCION Y EL PUERTO */
        if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
                error("ERROR on binding");

        /* SETEA LA CANTIDAD QUE PUEDEN ESPERAR MIENTRAS SE MANEJA UNA CONEXION */
        listen(sockfd, 5);

        /* SE BLOQUEA A ESPERAR UNA CONEXION */
        clilen = sizeof(cli_addr);
        newsockfd = accept(sockfd,
                           (struct sockaddr *) &cli_addr,
                           &clilen);

        /* DEVUELVE UN NUEVO DESCRIPTOR POR EL CUAL SE VAN A REALIZAR LAS COMUNICACIONES */
        if (newsockfd < 0)
                error("ERROR on accept");

        bzero(buffer, BUFFER_SIZE);

        /* LEE EL MENSAJE DEL CLIENTE */
        n = read(newsockfd, buffer, BUFFER_SIZE - 1);
        if (n < 0)
                error("ERROR reading from socket");

        printf("Receibed message with %d characters\n", n);

        if (verbose_flag) {
                printf("Message content:\n%s\n", buffer);
        }


        /* RESPONDE AL CLIENTE */
        n = write(newsockfd, "I got your message", 18);
        if (n < 0) error("ERROR writing to socket");

        return 0;
}


void
set_verbose_mode(int argc, char *argv[], int *verbose_flag_ptr)
{
        static int verbose_flag;
        int c;

        while (1)
        {
                static struct option long_options[] =
```

```
                {
                        {"verbose", no_argument, &verbose_flag, 1},
                        {"brief", no_argument, &verbose_flag, 0},
                        {0, 0, 0, 0}
                };
                /* getopt_long stores the option index here. */
                int option_index = 0;

                c = getopt_long(argc, argv, "vb",
                                long_options, &option_index);

                /* Detect the end of the options. */
                if (c == -1)
                        break;

                switch (c)
                {
                case 0:
                        /* If this option set a flag, do nothing else now. */
                        if (long_options[option_index].flag != 0)
                                break;
                        printf("option %s", long_options[option_index].name);
                        if (optarg)
                                printf(" with arg %s", optarg);
                        printf("\n");
                        break;
                case 'v':
                        verbose_flag = 1;
                        break;
                case 'b':
                        verbose_flag = 0;
                        break;
                case '?':
                        /* getopt_long already printed an error message. */
                        break;
                default:
                        exit(0);
                }
        }

        *verbose_flag_ptr = verbose_flag;
}
```

6.B. Ejercicio 2c

Modificación de los programas en C que permita verificar la llegada correcta de los datos.

6.B.1. Cliente

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

unsigned long djb2(unsigned char *str);

/*
```

```
 * En tiempo de compilación se puede definir esta macro con un valor numérico >= 0
 *
 */
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1000
#endif

/*
 * Error
 * -- Muestra un mensaje de error y finaliza la ejecución del programa.
 *
 * @param char * msg, Mensaje a mostrar
 *
 * @return void
 *
 */
void
error(char *msg)
{
        perror(msg);
        exit(0);
}

int
main(int argc, char *argv[])
{
        int sockfd, portno, n;
        struct sockaddr_in serv_addr;
        struct hostent *server;

        char buffer[BUFFER_SIZE];

        if (argc < 3)
        {
                fprintf(stderr, "usage %s hostname port\n", argv[0]);
                exit(0);
        }

        /* TOMA EL NUMERO DE PUERTO DE LOS ARGUMENTOS */
        portno = atoi(argv[2]);

        /* CREA EL FILE DESCRIPTOR DEL SOCKET PARA LA CONEXION */
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        /* AF_INET - FAMILIA DEL PROTOCOLO - IPV4 PROTOCOLS INTERNET */
        /* SOCK_STREAM - TIPO DE SOCKET */

        if (sockfd < 0)
                error("ERROR opening socket");

        /* TOMA LA DIRECCION DEL SERVER DE LOS ARGUMENTOS */
        server = gethostbyname(argv[1]);
        if (server == NULL)
        {
                fprintf(stderr, "ERROR, no such host\n");
                exit(0);
        }

        bzero((char *) &serv_addr, sizeof(serv_addr));
```

```
        serv_addr.sin_family = AF_INET;

        /* COPIA LA DIRECCION IP Y EL PUERTO DEL SERVIDOR A LA ESTRUCTURA DEL SOCKET */
        bcopy((char *) server->h_addr,
              (char *) &serv_addr.sin_addr.s_addr,
              server->h_length);
        serv_addr.sin_port = htons(portno);

        /* DESCRIPTOR - DIRECCION - TAMAÑO DIRECCION */
        if (connect(sockfd, (const struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
                error("ERROR connecting");

        *buffer = 'y';
        memset((buffer + 1), 'e', BUFFER_SIZE - 2);

        /* ENVIA UN MENSAJE AL SOCKET */
        n = write(sockfd, buffer, strlen(buffer));
        if (n < 0)
                error("ERROR writing to socket");

        //Envia el tamaño del dato
        size_t data_length = strlen(buffer);
        n = write(sockfd, &data_length, sizeof(data_length));

        //Envia el checksum
        unsigned long buffer_hash = djb2(buffer);
        n = write(sockfd, &buffer_hash, sizeof(buffer_hash));

        bzero(buffer, BUFFER_SIZE);

        /* ESPERA RECIBIR UNA RESPUESTA */
        n = read(sockfd, buffer, BUFFER_SIZE - 1);
        if (n < 0)
                error("ERROR reading from socket");

        printf("%s\n", buffer);

        return 0;
}

unsigned long
djb2(unsigned char *str)
{
        unsigned long hash = 5381;
        int c;

        while (c = *str++)
                hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

        return hash;
}
```

6.B.2. Servidor

```
/* A simple server in the internet domain using TCP
   The port number is passed as an argument */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

```c
#include <getopt.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>


unsigned long djb2(unsigned char *str);
void set_verbose_mode(int argc, char *argv[], int *verbose_flag_ptr);

/*
 * En tiempo de compilación se puede definir esta macro con un valor numérico >= 0
 *
 */
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1000
#endif


/*
 * Error
 * -- Muestra un mensaje de error y finaliza la ejecución del programa.
 *
 * @param char * msg, Mensaje a mostrar
 *
 * @return void
 *
 */
void
error(char *msg)
{
        perror(msg);
        exit(0);
}


int
main(int argc, char *argv[])
{
        static int verbose_flag;

        int sockfd, newsockfd, portno, clilen;
        char buffer[BUFFER_SIZE];
        struct sockaddr_in serv_addr, cli_addr;
        int n;

        if (argc < 2)
        {
                fprintf(stderr, "ERROR, no port provided\n");
                exit(1);
        }

        /* CREA EL FILE DESCRIPTOR DEL SOCKET PARA LA CONEXION */
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        /* AF_INET - FAMILIA DEL PROTOCOLO - IPV4 PROTOCOLS INTERNET */
        /* SOCK_STREAM - TIPO DE SOCKET */

        if (sockfd < 0)
                error("ERROR opening socket");
```

```
bzero((char *) &serv_addr, sizeof(serv_addr));
/* ASIGNA EL PUERTO PASADO POR ARGUMENTO */
/* ASIGNA LA IP EN DONDE ESCUCHA (SU PROPIA IP) */
portno = atoi(argv[1]);
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

set_verbose_mode(argc, argv, &verbose_flag);

/* Instead of reporting ‘--verbose’
and ‘--brief’ as they are encountered,
we report the final status resulting from them. */
if (verbose_flag)
        puts("verbose flag is set");

/* VINCULA EL FILE DESCRIPTOR CON LA DIRECCION Y EL PUERTO */
if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        error("ERROR on binding");

/* SETEA LA CANTIDAD QUE PUEDEN ESPERAR MIENTRAS SE MANEJA UNA CONEXION */
listen(sockfd, 5);

/* SE BLOQUEA A ESPERAR UNA CONEXION */
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd,
                   (struct sockaddr *) &cli_addr,
                   &clilen);

/* DEVUELVE UN NUEVO DESCRIPTOR POR EL CUAL SE VAN A REALIZAR LAS COMUNICACIONES */
if (newsockfd < 0)
        error("ERROR on accept");

bzero(buffer, BUFFER_SIZE);

/* LEE EL MENSAJE DEL CLIENTE */
n = read(newsockfd, buffer, BUFFER_SIZE - 1);
if (n < 0)
        error("ERROR reading from socket");

printf("Receibed message with %d characters\n", n);

if (verbose_flag) {
        printf("Message content:\n%s\n", buffer);
}

/* Get data length */
size_t data_length;
n = read(newsockfd, &data_length, sizeof(data_length));
if (n < 0) error("ERROR reading from socket");
printf("Received size: %lu\nSize sent by Client %lu\n", strlen(buffer), data_length);

/* Get client checksum */
unsigned long buffer_checksum;
n = read(newsockfd, &buffer_checksum, sizeof(buffer_checksum));
if (n < 0) error("ERROR reading from socket");
printf("Client Checksum: %lu\nServer checksum %lu\n", buffer_checksum, djb2(buffer));
```

```
        /* RESPONDE AL CLIENTE */
        n = write(newsockfd, "I got your message", 18);
        if (n < 0) error("ERROR writing to socket");

        return 0;
}


unsigned long
djb2(unsigned char *str)
{
        unsigned long hash = 5381;
        int c;

        while (c = *str++)
                hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

        return hash;
}


void
set_verbose_mode(int argc, char *argv[], int *verbose_flag_ptr)
{
        static int verbose_flag;
        int c;

        while (1)
        {
                static struct option long_options[] =
                {
                        {"verbose", no_argument, &verbose_flag, 1},
                        {"brief", no_argument, &verbose_flag, 0},
                        {0, 0, 0, 0}
                };
                /* getopt_long stores the option index here. */
                int option_index = 0;

                c = getopt_long(argc, argv, "vb",
                                long_options, &option_index);

                /* Detect the end of the options. */
                if (c == -1)
                        break;

                switch (c)
                {
                case 0:
                        /* If this option set a flag, do nothing else now. */
                        if (long_options[option_index].flag != 0)
                                break;
                        printf("option %s", long_options[option_index].name);
                        if (optarg)
                                printf(" with arg %s", optarg);
                        printf("\n");
                        break;
```

```
                case 'v':
                        verbose_flag = 1;
                        break;
                case 'b':
                        verbose_flag = 0;
                        break;
                case '?':
                        /* getopt_long already printed an error message. */
                        break;
                default:
                        exit(0);
                }
        }

        *verbose_flag_ptr = verbose_flag;
}
```

6.C. Ejercicio 2d

Modificación de los programas en C que permita verificar medir tiempo de conexión.

6.C.1. Cliente

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <getopt.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

double dwalltime(void);
unsigned long djb2(unsigned char *str);
void set_verbose_mode(int argc, char *argv[], int *verbose_flag_ptr);

/*
 * En tiempo de compilación se puede definir esta macro con un valor numérico >= 0
 *
 */
#ifndef BUFFER_SIZE
#define BUFFER_SIZE 1000
#endif

/*
 * Error
 * -- Muestra un mensaje de error y finaliza la ejecución del programa.
 *
 * @param char * msg, Mensaje a mostrar
 *
 * @return void
 *
 */
void
error(char *msg)
{
        perror(msg);
        exit(0);
}
```

```c
int
main(int argc, char *argv[])
{
        static int verbose_flag;

        int sockfd, portno, n;
        struct sockaddr_in serv_addr;
        struct hostent *server;

        char buffer[BUFFER_SIZE];

        if (argc < 3)
        {
                fprintf(stderr, "usage %s hostname port\n", argv[0]);
                exit(0);
        }

        /* TOMA EL NUMERO DE PUERTO DE LOS ARGUMENTOS */
        portno = atoi(argv[2]);

        /* CREA EL FILE DESCRIPTOR DEL SOCKET PARA LA CONEXION */
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        /* AF_INET - FAMILIA DEL PROTOCOLO - IPV4 PROTOCOLS INTERNET */
        /* SOCK_STREAM - TIPO DE SOCKET */

        if (sockfd < 0)
                error("ERROR opening socket");

        /* TOMA LA DIRECCION DEL SERVER DE LOS ARGUMENTOS */
        server = gethostbyname(argv[1]);
        if (server == NULL)
        {
                fprintf(stderr, "ERROR, no such host\n");
                exit(0);
        }

        set_verbose_mode(argc, argv, &verbose_flag);

        /* Instead of reporting ‘--verbose’
        and ‘--brief’ as they are encountered,
        we report the final status resulting from them. */
        if (verbose_flag)
                puts("verbose flag is set");

        bzero((char *) &serv_addr, sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;

        /* COPIA LA DIRECCION IP Y EL PUERTO DEL SERVIDOR A LA ESTRUCTURA DEL SOCKET */
        bcopy((char *) server->h_addr,
              (char *) &serv_addr.sin_addr.s_addr,
              server->h_length);
        serv_addr.sin_port = htons(portno);

        /* DESCRIPTOR - DIRECCION - TAMAÑO DIRECCION */
        if (connect(sockfd, (const struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
                error("ERROR connecting");
```

```
        *buffer = 'y';
        memset((buffer + 1), 'e', BUFFER_SIZE - 2);

        double timetick = dwalltime();

        /* ENVIA UN MENSAJE AL SOCKET */
        n = write(sockfd, buffer, strlen(buffer));
        if (n < 0)
                error("ERROR writing to socket");

        //Envia el tamaño del dato
        size_t data_length = strlen(buffer);
        n = write(sockfd, &data_length, sizeof(data_length));

        //Envia el checksum
        unsigned long buffer_hash = djb2(buffer);
        n = write(sockfd, &buffer_hash, sizeof(buffer_hash));

        bzero(buffer, BUFFER_SIZE);

        /* ESPERA RECIBIR UNA RESPUESTA */
        n = read(sockfd, buffer, BUFFER_SIZE - 1);
        if (n < 0)
                error("ERROR reading from socket");

        if (verbose_flag) {
                printf("%s\n", buffer);
                printf("Time in seconds: %fs\n", dwalltime() - timetick);
        } else {
                printf("%f\n", dwalltime() - timetick);
        }

        return 0;
}


unsigned long
djb2(unsigned char *str)
{
        unsigned long hash = 5381;
        int c;

        while (c = *str++)
                hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

        return hash;
}


double
dwalltime(void)
{
        double sec;
        struct timeval tv;

        gettimeofday(&tv, NULL);
        sec = tv.tv_sec + tv.tv_usec/1000000.0;
        return sec;
```

```
}


void
set_verbose_mode(int argc, char *argv[], int *verbose_flag_ptr)
{
        static int verbose_flag;
        int c;

        while (1)
        {
                static struct option long_options[] =
                {
                        {"verbose", no_argument, &verbose_flag, 1},
                        {"brief", no_argument, &verbose_flag, 0},
                        {0, 0, 0, 0}
                };
                /* getopt_long stores the option index here. */
                int option_index = 0;

                c = getopt_long(argc, argv, "vb",
                                long_options, &option_index);

                /* Detect the end of the options. */
                if (c == -1)
                        break;

                switch (c)
                {
                case 0:
                        /* If this option set a flag, do nothing else now. */
                        if (long_options[option_index].flag != 0)
                                break;
                        printf("option %s", long_options[option_index].name);
                        if (optarg)
                                printf(" with arg %s", optarg);
                        printf("\n");
                        break;
                case 'v':
                        verbose_flag = 1;
                        break;
                case 'b':
                        verbose_flag = 0;
                        break;
                case '?':
                        /* getopt_long already printed an error message. */
                        break;
                default:
                        exit(0);
                }
        }

        *verbose_flag_ptr = verbose_flag;
}
```

6.C.2. Servidor

La implementación del servidor no se modificó respecto de la presente en la sección 6.B.2.